

A Logo Interpreter

Introduction

Suppose you were marooned on a desert island, with only your Atari and an assembler/editor cartridge. If you wanted to use Logo, you would have to write it yourself. How would you go about writing a computer language? You would have to write a program that runs the language. It is possible to write such a program, with some simplifications, in Logo itself.

Logo is an *interpreted* language. When you run your programs, Logo reads through them one instruction line at a time and executes each instruction in the line before proceeding to the next line. This is called interpreting a program.

Once you grasp the basic principles of interpreter design and operation, you could write an interpreter for any computer language, not just Logo. And you could write your interpreter in another language, like assembly language.

This project is about writing an interpreter for Logo in Atari Logo. We'll call this interpreter MLogo(for micro-Logo), to distinguish it from Atari Logo, which is an interpreter written in Atari machine language.

How to Use MLogo

To use MLogo, first initialize it (INIT), then start it (LOGO).

MLogo will prompt you for input with a ? in inverse video.

MLogo has fewer primitives than Atari Logo; among them are some list and arithmetic operations and some turtle commands.

```
?PRINT SUM 3 4
7
?PRINT FIRST BF "WALLABEE
A
?FD 100
?
```

You can write procedures in MLogo.

```
?TO POLY :SIDE :ANGLE
>FD :SIDE
>RT :ANGLE
>POLY :SIDE :ANGLE
>END
POLY DEFINED
```

MLogo is different from Atari Logo in some ways. MLogo doesn't care whether a line outputs or not. If you type:

```
SUM 3 4
```

By Jim Davis and Ed Hardebeck. An earlier version of this project was written by Henry Minsky.

the value 7 is just ignored. In Atari Logo you would get the error message:

```
YOU DON'T SAY WHAT TO DO WITH 7
```

MLogo doesn't have the STOP primitive. Every line of a user procedure is executed. It also doesn't have OP. The value of a user procedure is the value of the last line in it.

```
?TO GREET :WHO
>SE "HELLO :WHO
>END
GREET DEFINED
?PRINT GREET "ARTHUR
HELLO ARTHUR
```

If you typed this to Atari Logo, you'd get an error:

```
YOU DON'T SAY WHAT TO DO WITH [HELLO ARTHUR] IN GREET
```

Another difference is that all variables start with the empty list as their value.

```
?SHOW :NOVAL
[]
```

In Atari Logo, you'd get an error.

```
NOVAL HAS NO VALUE
```

If you try to use an undefined procedure in MLogo, you get a mysterious error message, then MLogo "crashes."

```
?ZIPPER 3
$ZIPPER HAS NO VALUE IN FSYMEVAL
```

After MLogo crashes, you are once again talking to Atari Logo. You must restart MLogo.

You may notice other differences as well. The reason for these differences is that it's difficult to implement Logo completely.

Now that you've had a chance to use MLogo and know what it does, we'll explain how it does it. The discussion, however, omits many details about interpreters.* Throughout this explanation we use the technical terms usually used by Logo implementors for describing Logo interpreters.

Interpretation Happens a Line at a Time

The structure of MLogo resembles that of Atari Logo. The normal action of Logo is to repeatedly type a prompt (?), read a line from the keyboard,

*For more information see *Structure and Interpretation of Computer Programs* by Gerald J. Sussman and Harold Abelson, MIT Press and McGraw-Hill, 1984.

PROGRAMMING IDEAS

and *evaluate* it. The top-level loop of MLogo is:

```
TO LOGOLOOP
  IGNORE EVLINE GET.LINE
  LOGOLOOP
END
```

The output of GET.LINE is a list of what the user typed.

EVLINE accepts a line as its input and carries out whatever instructions the line contains (for example, moves the turtle, prints a sentence, and so on).

```
TO EVLINE :LINE
  OP EVLINE1 "$NOVALUE
END

TO EVLINE1 :VALUE
  IF EMPTY? :LINE [OP :VALUE]
  OP EVLINE1 EVAL NEXT.ITEM
END
```

EVLINE1 does the actual evaluation of the instructions of a line. The easiest way to understand it is to look first at its last line.

The operation NEXT.ITEM removes the first item from the variable LINE and outputs it. Each call to NEXT.ITEM removes one item and outputs it. In this way each item is inspected in turn.

```
TO NEXT.ITEM
  OP NEXT.ITEM1 FIRST :LINE
END

TO NEXT.ITEM1 :FIRST
  MAKE "LINE BF :LINE
  OP :FIRST
END
```

EVAL takes an item, decides what kind of thing it is, and evaluates it to get its value. The value output from this call to EVAL is the input to EVLINE1 when it recurses. If there's nothing left on the line, this is the value to output. Otherwise, there's another instruction on the line.

When EVLINE calls EVLINE1, none of the line has been evaluated. If it should turn out that the line has no instructions (a blank line), then there is no value to output. \$NOVALUE is just a default value.

Here's an example of how this recursion works. Suppose you type

```
FD 60 RT 90
```

to MLogo. LOGOLOOP calls EVLINE with the list [FD 60 RT 90] as input. NEXT.ITEM outputs FD, and :LINE is [60 RT 90]. FD is passed to EVAL for evaluation.

In evaluating FD, the number 60 would be removed from the line (it is an input to FD). When EVAL stops, the value of :LINE is [RT 90]. Since this is not an empty list, evaluation would continue.

The Rules of EVAL

The value of an item is determined by these rules.

- The value of a list is just the list.
- The value of a number is just the number.
- The value of a quoted word is the word itself without the quote.
- The value of a word prefaced with a colon (called "dots") is the value of the variable.
- Otherwise the word is the name of a procedure to call. Inputs to the procedure appear after the name of the procedure.

EVAL looks at an item to see what type it is, then carries out the appropriate rule.

How EVAL *Carries Out Its Rules*

```
TO EVAL :ITEM
IF LISTP :ITEM [OUTPUT :ITEM]
IF NUMBERP :ITEM [OUTPUT :ITEM]
IF QUOTED? :ITEM [OUTPUT UNQUOTE :ITEM]
IF DOTTED? :ITEM [OUTPUT GET.VARIABLE.VALUE UNDOT :ITEM]
OUTPUT EVAL.CALL FSYMEVAL :ITEM
END
```

EVAL's first test is for a list. If the item isn't a list, it must be a word. The remaining tests all assume the item is some kind of word and don't include WORDP as part of the test.

The predicate QUOTED? tests whether its input is a quoted word.

```
TO QUOTED? :WORD
OUTPUT EQUALP FIRST :WORD ""
END
```

The operation UNQUOTE removes the quote and outputs the word.

```
TO UNQUOTE :WORD
OUTPUT BF :WORD
END
```

In Logo a colon ("dots") before a word is a request for the value of a variable. The predicate DOTTED? checks this case.

```
TO DOTTED? :WORD
OUTPUT EQUALP FIRST :WORD ":"
END
```

The procedure UNDOT outputs the word with the dots removed.

```
TO UNDOT :WORD
OUTPUT BF :WORD
END
```

GET.VARIABLE.VALUE outputs the value of a variable.

PROGRAMMING IDEAS

```

TO GET.VARIABLE.VALUE :WORD
IF BOUND? :WORD [OUTPUT SYMEVAL :WORD]
OUTPUT []
END

```

The predicate `BOUND?` checks whether the word has been assigned a value. If it has one, `SYMEVAL` outputs it, otherwise the value is `[]`. We'll explain more about this later on.

Evaluating a Procedure Call

To evaluate a procedure call, we have to know some things about the procedure being called, such as how many inputs it has and whether it's a primitive or a user procedure. Information about a procedure is kept in the *definition* of the procedure. We'll describe definitions in detail later. For now, we'll just say that the operation `FSYMEVAL` outputs the definition and leave it at that.

When `EVAL` wishes to evaluate a procedure, it passes the definition of the procedure to `EVAL.CALL`.

```

TO EVAL.CALL :DEFINITION
OP APPLY :DEFINITION EVAL.ARGS NARGS :DEFINITION
END

```

`APPLY` actually runs the procedure. It takes two inputs. The first is the definition of a procedure, the second is a list of values for the inputs. It causes the procedure to "do its thing," whatever that is, and `APPLY` outputs whatever the procedure outputs.

Before we can run the procedure, we have to get the values of its inputs. We usually refer to inputs as *arguments* (or *args*, for short).

The operation `NARGS` outputs the number of arguments this procedure expects. `NARGS` extracts this from the definition. (We'll see `NARGS` later.) This number is the input to `EVAL.ARGS`. `EVAL.ARGS` takes these inputs from the line being evaluated and evaluates each one, returning a list of the values.

To simplify MLogo, everything outputs. If commands were allowed in MLogo, as they are in Atari Logo, `EVAL.CALL` would have to know if an output was expected and make the proper complaint if an output was missing or an unexpected output showed up.

Inputs Require Recursive Evaluation

```

TO EVAL.ARGS :NARGS
IF EQUALP :NARGS 0 [OP []]
OUTPUT FPUT EVAL.NEXT.ITEM EVAL.ARGS :NARGS - 1
END

```

`EVAL.ARGS` calls `NEXT.ITEM` to get the next item in the line being evaluated and makes a recursive call to `EVAL` to evaluate this item.

Here's an example. Suppose we type:

```
MAKE "DOGS 3
PRINT :DOGS
```

to MLogo. Our example begins after the MAKE, when evaluating the call to PRINT.

EVLINE gets [PRINT :DOGS].

NEXT.ITEM outputs PRINT.

EVAL gets PRINT and decides it's the first word of a procedure call, so it calls EVAL.CALL with the definition.

EVAL.CALL calls NARGS to get the number of arguments that PRINT wants, and passes this to EVAL.ARGS.

EVAL.ARGS gets 1 as input, so it calls NEXT.ITEM, which outputs :DOGS. EVAL.ARGS calls EVAL to evaluate it.

EVAL gets :DOGS as input. This is a dotted word so GET.VARIABLE.VALUE is called with :DOGS. It outputs the value DOGS, which is 3. EVAL outputs 3.

EVAL.ARGS recurses. :NARGS - 1 is 0.

EVAL.ARGS is called with 0, so it outputs the empty list.

EVAL.ARGS outputs FPUT 3 [] to EVAL.CALL.

EVAL.CALL calls APPLY with the definition and the list of values returned by EVAL.ARGS, in this case [3].

APPLY invokes PRINT with an input of 3. Whatever APPLY outputs is what EVAL.CALL outputs.

EVAL.CALL returns to EVAL, which returns to EVLINE1.

Before we show how APPLY works, we'll give some details of procedure definitions.

Procedure Definitions

Both primitives and user procedures have definitions. Their definitions have some features in common and some differences.

In the remainder of this discussion, we refer to a primitive as an *sfun* (System FUNction), pronounced "ess-fun." Likewise we refer to a user procedure as a *ufun* (User FUNction), pronounced "you-fun." These are the terms usually used by Logo implementors.

Procedure definitions are kept in lists.

The first item in the list is the word SFUN or UFUN. The predicate SFUN? distinguishes sfuns from ufuncs by inspecting this item.

```
TO SFUN? :DEFINITION
OP EQUALP FIRST :DEFINITION "SFUN
END
```

The second item is the number of inputs the procedure expects (this may be zero). The operation NARGS outputs this number.

```
TO NARGS :DEFINITION
OP FIRST BF :DEFINITION
END
```

The remaining items of the list differ for the two types of procedures. We'll show you the rest of an sfun definition now and take up ufuncs later.

PROGRAMMING IDEAS

The third and final item in an sfun definition is the name of the Atari Logo procedure that implements the MLogo primitive.

The operation SFUN.FUNC outputs this procedure.

```
TO SFUN.FUNC :DEFINITION
  OUTPUT FIRST BF BF :DEFINITION
END
```

If you print the names in the Logo workspace, you'll see definitions for all the MLogo primitives. All the definitions are in words beginning with \$.

```
?SHOW :$PRINT
[SFUN 1 %PRINT]
?SHOW NARGS :$PRINT
1
?SHOW SFUN.FUNC :$PRINT
%PRINT
?SHOW :$SUM
[SFUN 2 SUM]
```

Sometimes an MLogo primitive is implemented directly by an Atari Logo primitive (for example, SUM), and sometimes by a procedure (%PRINT).

The operation MAKE.SFUN.DEF makes a definition for an sfun.

```
TO MAKE.SFUN.DEF :NARGS :FUNC
  OUTPUT (SE "SFUN :NARGS :FUNC)
END
```

Now we can finish discussing the evaluation of a procedure call.

APPLY *Evaluates a Procedure Call*

The first input to APPLY is the definition of a procedure to evaluate. The second input is a list of input values for that procedure. Sfun and ufun are evaluated differently.

```
TO APPLY :DEFINITION :VALUES
  IF SFUN? :DEFINITION [OP APPLY.SFUN :DEFINITION :VALUES]
  OP APPLY.UFUN :DEFINITION :VALUES
END
```

The command APPLY.SFUN applies an sfun to its inputs by building a list as input for RUN.

```
TO APPLY.SFUN :DEF :VALUES
  OUTPUT RUN SE SFUN.FUNC :DEF ( QUOTIFY :VALUES )
END
```

Suppose you typed the following to MLogo:

```
MAKE "WHO "LOWELL
PRINT :WHO
```

While evaluating the call to PRINT, APPLY.SFUN would get the inputs [SFUN 1 %PRINT] (the definition of PRINT) and LOWELL (the value of the variable WHO).

Recall that SFUN.FUNC outputs the procedure that implements the sfun. In our example it will output %PRINT.

APPLY.SFUN would call RUN with the input [%PRINT "LOWELL"]. RUN would call %PRINT on behalf of MLogo and output whatever it output.

Here's the MLogo sfun PRINT.

```
TO %PRINT :ARG
  PRINT :ARG
  OUTPUT :ARG
END
```

The operation QUOTIFY puts a quote in front of words that need it.

```
TO QUOTIFY :VALS
  IF EMPTY? :VALS [OUTPUT []]
  OUTPUT FPUT QUOTIFY1 FIRST :VALS QUOTIFY BF :VALS
END
```

```
TO QUOTIFY1 :VAL
  IF NUMBERP :VAL [OP :VAL]
  IF WORDP :VAL [OP WORD " " :VAL]
  OUTPUT :VAL
END
```

Variable Values

The values of MLogo variables are stored in Atari Logo variables with slightly "funny" names. (This is useful for learning about how MLogo works. You can stop it and print out names. You can easily spot all MLogo variables by their names.)

The operation VSYM makes these names by adding a # to the front of the name. (The name VSYM stands for Variable SYMbol.)

```
TO VSYM :WORD
  OP WORD "# :WORD
END
```

The command SET sets the value of an MLogo word, and SYMEVAL gets the value of an MLogo word. They both use VSYM to get the name of the word to use. VSYM *translates* from an MLogo name to an Atari Logo name.

```
TO SET :SYM :VAL
  MAKE VSYM :SYM :VAL
END
```

```
TO SYMEVAL :SYM
  OP THING VSYM :SYM
END
```


PROGRAMMING IDEAS

The predicate `BOUND?` tells whether there is a value for the word.

```
TO BOUND? :WORD
OP NAMEP VSYM :WORD
END
```

A second reason to use “funny” names for MLogo variables is that otherwise an MLogo user might set a variable with the same name as one used in the MLogo program itself. The results would be very strange. Adding the character guarantees that the names will never be the same.

Using a scheme like the one for variables, the definition of a procedure is kept in a variable whose name is the name of the procedure with a “\$” prefix. The operation `FSYM` (Function SYMbol) outputs the Logo variable for the definition of the MLogo procedure.

```
TO FSYM :WORD
OP WORD "$ :WORD
END
```

The command `FSET` sets the definition of a procedure, and the operation `FSYMEVAL` outputs the definition of a procedure.

```
TO FSET :SYMBOL :DEF
MAKE FSYM :SYMBOL :DEF
END
```

```
TO FSYMEVAL :NAME
OUTPUT THING FSYM :NAME
END
```

How Sfun Are Defined

The primitives we implemented are all very similar to familiar Logo primitives. In some cases we could call Logo primitives directly. But because every MLogo *sfun* must output, we had to write small Atari Logo procedures for those that don’t output. These procedures call the *sfun*, then output a value. The value may just be `TRUE`.

```
TO %PRINT :ARG
PRINT :ARG
OUTPUT :ARG
END
```

```
TO %MAKE :SYM :VAL
SET.VARIABLE.VALUE :SYM :VAL
OUTPUT :VAL
END
```

```
TO %FD :N
FD :N
OP "TRUE
END
```

DEF.SFUN defines an sfun, that is, it associates the name of an sfun with the definition.

```
TO DEF.SFUN :NAME :NARGS :FUNC
FSET :NAME MAKE.SFUN.DEF :NARGS :FUNC
END
```

All sfun procedures' names begin with a percent sign to distinguish them from procedures that are part of MLogo itself. This makes it easy to spot all the MLogo sfuns in the workspace (except those implemented directly by Atari Logo primitives).

Ufun Definitions Include Arglist and Body

Like sfuns, unfuns have a definition, but the definition is slightly different. A user procedure consists of an *arglist* and a *body*. The arglist is a list of the input variables for the unfun. The body is a list of the lines of the procedure. Like sfuns, unfun definitions are lists.

If we had defined SQUARE by

```
TO SQUARE :N
PRINT :N
PRODUCT :N :N
END
```

... then the definition would be

```
?SHOW FSYMEVAL "SQUARE
[UFUN 1 [N] [[PRINT :N][PRODUCT :N :N]]]
```

The arglist is [N], the body is [[PRINT :N] [PRODUCT :N :N]]. Remember that MLogo unfun definitions are stored by the interpreter as Atari Logo *variables*, not as Atari Logo *procedures*.

The operation MAKE.UFUN.DEF makes a definition for a unfun. The operation UFUN.ARGLIST outputs the arglist from the definition, and the operation UFUN.BODY extracts the unfun body from the definition.

```
TO MAKE.UFUN.DEF :ARGS :BODY
OP (SE "UFUN COUNT :ARGS LIST :ARGS :BODY)
END
```

```
TO UFUN.ARGLIST :DEFINITION
OUTPUT FIRST BF BF :DEFINITION
END
```

```
TO UFUN.BODY :DEFINITION
OUTPUT FIRST BF BF BF :DEFINITION
END
```


Evaluating a Ufun Means Evaluating the Lines of Its Body

An sfun is a primitive, but a ufun body is a collection of lines, each requiring evaluation itself.

```
TO EVAL.BODY :LINES
OP EVAL.BODY1 "$NOVALUE :LINES
END

TO EVAL.BODY1 :VALUE :LINES
IF EMPTY? :LINES [OP :VALUE]
OP EVAL.BODY1 EVLINE FIRST :LINES BF :LINES
END
```

EVAL.BODY1 does the actual evaluation of the lines of the body. The value of the ufun is the value of the last line evaluated.

EVAL.BODY1 recurses in the same way EVLINE does. To understand it, look at the recursive call first. Each time EVAL.BODY1 recurses its first input is the value from evaluating the previous line. When the last line is evaluated, this is the value to output.

When EVAL.BODY1 is first called (from EVAL.BODY), it is passed \$NOVALUE as a first input. When first called, EVAL.BODY1 has yet to evaluate a line, so there is no value to output from the ufun. If the ufun body is empty, then \$NOVALUE is output. Otherwise there is at least one line to evaluate. EVAL.BODY1 evaluates the first line in the body, and recurses with this value and the remainder of the lines.

Ufuns Have Inputs with Names

Ufuns can have inputs. The title line (and therefore the arglist) of a ufun lists a set of variables that hold the inputs to the ufun. While a ufun is being evaluated, it can find its inputs in these variables.

For example, if you have the procedure:

```
TO GREET :WHO
PRINT SE "HELLO :WHO
END
```

and you type:

```
GREET "PHIL
```

Logo (either Atari Logo or MLogo) responds:

```
HELLO PHIL
```

Logo acts as if the value of :WHO had been set by MAKE before any of the instructions were evaluated. The effect is like what you could get by

```
?MAKE "WHO "PHIL
?PRINT SE "HELLO :PHIL
```

The difference is that after the ufun GREET is finished, the variable WHO has the same value it had before. Try it yourself if you don't already know this.

```
?MAKE "WHO [BAKED HAM]
?GREET "BOB
HELLO BOB
?SHOW :WHO
[BAKED HAM]
?MAKE "WHO "BOB
?PRINT SE "HELLO :WHO
HELLO BOB
?SHOW :WHO
BOB
```

Before EVAL.BODY can do its work, the previous values of certain variables must be saved before those variables receive new values. The input variables hold their values only for the duration of evaluation of the ufun, and then they have their old values restored.

The process of setting and restoring of values is referred to as *binding* the variables. Making binding work properly is one of the most difficult parts of writing an interpreter.

APPLY.UFUN is called by APPLY to evaluate a ufun. See APPLY.SFUN for comparison.

```
TO APPLY.UFUN :DEF :VALUES
BIND.ARGS UFUN.ARGLIST :DEF :VALUES
OP CLEANUP EVAL.BODY UFUN.BODY :DEF
END
```

BIND.ARGS saves the old values of variables in the arglist, then sets the new values.

EVAL.BODY evaluates the forms of the body and outputs a value.

CLEANUP restores variables to their previous values. It outputs the value of the ufun.

How Binding Is Implemented

BIND.ARGS does two things. It saves old values and it sets new ones. It cooperates with CLEANUP, which restores the old values. These two procedures cooperate through the global variable BIND.STACK, which is where BIND.ARGS saves the values and CLEANUP finds them.

```
TO BIND.ARGS :ARGLIST :VALUES
PUSH "BIND.STACK BIND.FRAME :ARGLIST
SET.ARGS :ARGLIST :VALUES
END
```

The saved values are referred to collectively as a *bind frame*. The operation BIND.FRAME builds a bind frame for the variables in the arglist. BIND.ARGS uses PUSH to save this frame in the shared variable BIND.STACK, then calls SET.ARGS to set the new values.

PROGRAMMING IDEAS

```

TO SET.ARGS :NAMES :VALUES
IF EMPTY? :NAMES [STOP]
SET FIRST :NAMES FIRST :VALUES
SET.ARGS BF :NAMES BF :VALUES
END

```

SET.ARGS simply recurses through the argument list and the values. There is a one-to-one correspondence between the argument list and the values list. For each input there is a value.

After a ufun is evaluated it outputs a value, and this is the value that APPLY.UFUN should output as the value of the ufun it was asked to apply. But first the bound variables must be unbound. This is the purpose of CLEANUP.

```

TO CLEANUP :VALUE
UNBIND
OP :VALUE
END

```

CLEANUP's input is the output of the ufun. It holds onto this value while UNBIND undoes the binding, then returns the held value. UNBIND is explained later.

A bind frame enables the interpreter to restore the values of the input variables of a single ufun call. But a ufun can call other ufuncs. We need one bind frame for each ufun call. There will be as many bind frames as the depth of calling. Bind frames are created as calls occur and cleaned up as the call returns. The most recently added frame is always the one to clean up.

We need to keep track of all these bind frames and ensure we bind and unbind in the same order calls and returns are made. To do this, we use a *stack*.

The Concept of a Stack

A stack is a method of arranging data. You can think of it as a pile of papers on a desk. Only the topmost sheet is visible (if the stack is neat) because it covers the others. If you add another sheet to the pile, it becomes the topmost. You can only touch the top sheet. If you remove it, a new top sheet is exposed.

This order of accessing is sometimes referred to as "Last In, First Out," because the last item added to the stack is the first one that can be removed.

Stacks Are Implemented by Lists

Most computers have machine instructions to implement stacks. But since we wrote MLogo in Logo and not in machine language, we had to implement stacks. We decided to use Logo lists to hold stacks, and to put the top of the stack at the front of the list so that we could use FIRST to get the top item on the stack and FPUT to add a new one.

PUSH puts something on the top of a stack. It takes two inputs. The first is the name of the word containing the stack, the second is the item to add to the stack.


```

TO PUSH :STACK :ITEM
MAKE :STACK FPUT :ITEM THING :STACK
END

```

PUSH makes a new list by adding the item to the old contents of the stack. This new list is assigned to the variable holding the stack.

The operation POP outputs the top value on the stack. This value is removed from the stack.

```

TO POP :STACK
OP POP1 :STACK THING :STACK
END

```

```

TO POP1 :STACK :LIST
MAKE :STACK BF :LIST
OP FIRST :LIST
END

```

The input to the operation POP is the variable holding the stack. THING of this variable outputs its value—the list holding the stack. The first item in the list is the item to output. Before outputting it, POP1 sets the stack variable to hold the BF of the list, thus removing the top item from the stack.

This example shows how stacks work.

```

?MAKE "STACK []
?PUSH "STACK 9
?SHOW :STACK
[9]
?PUSH "STACK 5
?PUSH "STACK 2
?SHOW :STACK
[2 5 9]
?PRINT POP "STACK
2
?SHOW :STACK
[5 9]

```

Bind Frame in Detail

A bind frame is a list of bindings. Each binding is a list of a name and a value. The name is the name of a variable that must be saved, and the value is the value it had at the time it was saved.

A typical bind frame might be

```
[[A 3][NAME [JAMES ALLEN]]]
```

This bind frame is holding two variable bindings, for A and NAME.

BIND.FRAME makes a bind frame. Its input is the argument list of a ufun. Each input is a variable whose value must be saved.

```

TO BIND.FRAME :ARGLIST
IF EMPTY? :ARGLIST [OP []]
OP FPUT BIND.ARG FIRST :ARGLIST BIND.FRAME BF :ARGLIST
END

```

PROGRAMMING IDEAS

`BIND.FRAME` recurses through the list, collecting one binding for each variable. The operation `BIND.ARG` makes a binding for a variable. It outputs a list of the variable name and the current value of the variable.

```
TO BIND.ARG :NAME
OP LIST :NAME GET.VARIABLE.VALUE :NAME
END
```

`UNBIND` pops a single frame off the stack and passes it to `UNBIND.ARGS`, which acts like `BIND.FRAME` in reverse, restoring each saved value in the frame.

```
TO UNBIND
UNBIND.ARGS POP "BIND.STACK
END
```

```
TO UNBIND.ARGS :FRAME
IF EMPTY? :FRAME [STOP]
UNBIND.ARG FIRST :FRAME
UNBIND.ARGS BF :FRAME
END
```

```
TO UNBIND.ARG :PAIR
SET FIRST :PAIR FIRST BF :PAIR
END
```

The interpreter's top-level procedure, `LOGO`, initializes `BIND.STACK` to hold an empty list.

```
TO LOGO
MAKE "BIND.STACK []
LOGOLOOP
END
```

The Sfun TO Is Harder Than Others

In Logo the primitive `T0` treats its inputs differently from all other `sfuns`. It does not evaluate them. The first input to `T0` is the name of the procedure to define. The rest of the inputs are the names of the inputs of the procedure being defined. These are written with dots to remind you that they are the inputs.

```
?TO SQUARE :A
>PRODUCT :A :A
>END
SQUARE DEFINED
```

`T0` manages the trick of not evaluating its inputs by lying to the evaluator about its number of inputs. It says it takes none but then goes and takes them off `LINE` (where the current line is kept) by itself. `EVAL.ARGS` evaluates the arguments as it collects them. This trick also lets `T0` take as many arguments as are present on the line.

The `T0` definition is:


```
[SFUN 0 %T0]
```

The procedures that implement it are

```
TO %T0
OP T01 NEXT.ITEM GATHER.ARGS
END

TO T01 :NAME :ARGLIST
DEF.UFUN :NAME :ARGLIST READ.BODY
PRINT (SE :NAME "DEFINED)
OP "TRUE
END
```

The GATHER.ARGS operation pops the input names directly off LINE and removes the dots.

```
TO GATHER.ARGS
IF EMPTY? :LINE [OP []]
OP FPUT UNDOT NEXT.ITEM GATHER.ARGS
END
```

DEF.UFUN takes as inputs the name of the procedure to define, a list of its arguments, and its body, which is a list of the lines that make up the procedure.

```
TO DEF.UFUN :NAME :ARGS :BODY
FSET :NAME MAKE.UFUN.DEF :ARGS :BODY
END
```

MAKE.UFUN.DEF makes the actual definition. We have already seen it. READ.BODY reads an entire ufun body, prompting with > before reading each line.

```
TO READ.BODY
OP READ.BODY1 READ.LINE
END
```

```
TO READ.BODY1 :LINE
IF EQUALP :LINE [END] [OP []]
OP FPUT :LINE READ.BODY1 READ.LINE
END
```

READ.BODY1 recurses, reading a line each time, until it gets a line END.

Reading Things You Type

Both T0 and the LOGOLOOP need to get typein from the user. They don't want empty lines as input. Each has its own prompt character. They can both share INPUT.LINE.

```
TO GET.LINE
OP INPUT.LINE "¿
END
```

PROGRAMMING IDEAS

```

TO READ.LINE
OP INPUT.LINE ">
END

TO INPUT.LINE :PROMPT
TYPE :PROMPT
OP INPUT.LINE1 RL
END

TO INPUT.LINE1 :INPUT
IF NOT EMPTY? :INPUT [OP :INPUT]
OP INPUT.LINE :PROMPT
END

```

INPUT.LINE1 recurses until the user types a line that isn't empty.

Some Improvements

Here are some modifications to MLogo to make it more like Atari Logo. We didn't include them in MLogo because we wanted to keep it simple to explain. If you want to have these extra features, you can type in the following procedures.

First, a synonym for PRINT.

```
DEF.SFUN "PR 1 "%PRINT
```

Here's the sfun P0:

```

TO %P0 :NAME
TYPE SE "TO :NAME
P0.ARGS UFUN.ARGLIST FSYPEVAL :NAME
P0.BODY UFUN.BODY FSYPEVAL :NAME
OP :NAME
END

TO P0.ARGS :ARGLIST
IF EMPTY? :ARGLIST [PRINT [] STOP]
TYPE "\
TYPE " :
TYPE FIRST :ARGLIST
P0.ARGS BF :ARGLIST
END

TO P0.BODY :LINES
IF EMPTY? :LINES [PR "END STOP]
PRINT FIRST :LINES
P0.BODY BF :LINES
END

```

P0 is by far the longest sfun yet, because there is no useful Atari Logo primitive for it. The Atari Logo primitive P0 prints out Atari Logo user procedures, which are not stored like MLogo user procedures.

The procedure P0.ARGS prints each word in the argument list,

preceded by a space and a colon. (In the second line of P0.ARG5, a space appears after the backslash even though you can't tell from this listing.)

To add OP to MLogo we have to change EVAL.BODY and EVAL.BODY1. As is, each line is always evaluated. By adding a flag variable OP? we can cause evaluation to stop.

Here's the sfun OP.

```
TO %OP :VALUE
MAKE "OP :VALUE
MAKE "OP? "TRUE
OP "TRUE
END
```

```
DEF.SFUN "OP 1 "%OP
```

The input to OP is the value to output. This value is stored in the variable OP for reference by the evaluator. %OP sets the flag OP?, which causes the evaluation of the body to stop.

We have to modify the evaluator to check these flags.

```
TO EVAL.BODY :LINES
OP EVAL.BODY1 :LINES "FALSE
END

TO EVAL.BODY1 :LINES :OP?
IF EMPTY? :LINES [OP "$NOVALUE]
IGNORE EVLINE FIRST :LINES
IF :OP? [OP :OP]
OP EVAL.BODY1 BF :LINES "FALSE
```

PROGRAM LISTING

```
TO LOGO
MAKE "BIND.STACK []
LOGOLOOP
END
```

```
TO LOGOLOOP
IGNORE EVLINE GET.LINE
LOGOLOOP
END
```

```
TO EVLINE :LINE
OP EVLINE1 "$NOVALUE
END
```

```
TO EVLINE1 :VALUE
IF EMPTY? :LINE [OP :VALUE]
OP EVLINE1 EVAL NEXT.ITEM
END
```

```
TO NEXT.ITEM
OP NEXT.ITEM1 FIRST :LINE
END
```

```
TO NEXT.ITEM1 :FIRST
MAKE "LINE BF :LINE
OP :FIRST
END
```

```
TO EVAL :ITEM
IF LISTP :ITEM [OUTPUT :ITEM]
IF NUMBERP :ITEM [OUTPUT :ITEM]
IF QUOTED? :ITEM [OUTPUT UNQUOTE ►
:ITEM]
IF DOTTED? :ITEM [OUTPUT ►
GET.VARIABLE.VALUE UNDOT :ITEM]
OUTPUT EVAL.CALL FSYMEVAL :ITEM
END
```

```
TO QUOTED? :WORD
OUTPUT EQUALP FIRST :WORD ""
END
```

```
TO UNQUOTE :WORD
OUTPUT BF :WORD
END
```

```
TO DOTTED? :WORD
  OUTPUT EQUALP FIRST :WORD "
END
```

```
TO UNDOT :WORD
  OUTPUT BF :WORD
END
```

```
TO GET.VARIABLE.VALUE :WORD
  IF BOUND? :WORD [OUTPUT SYMEVAL :WORD]
  OUTPUT []
END
```

```
TO VSYM :WORD
  OP WORD "# :WORD
END
```

```
TO SET :SYM :VAL
  MAKE VSYM :SYM :VAL
END
```

```
TO SYMEVAL :SYM
  OP THING VSYM :SYM
END
```

```
TO BOUND? :WORD
  OP NAMEP VSYM :WORD
END
```

```
TO FSYM :WORD
  OP WORD "$ :WORD
END
```

```
TO FSYMEVAL :NAME
  OUTPUT THING FSYM :NAME
END
```

```
TO FSET :SYMBOL :DEF
  MAKE FSYM :SYMBOL :DEF
END
```

```
TO EVAL.CALL :DEFINITION
  OP APPLY :DEFINITION EVAL.ARGS NARGS ►
  :DEFINITION
END
```

```
TO MAKE.SFUN.DEF :NARGS :FUNC
  OUTPUT (SE "SFUN :NARGS :FUNC)
END
```

```
TO MAKE.UFUN.DEF :ARGS :BODY
  OP (SE "UFUN COUNT :ARGS LIST :ARGS ►
  :BODY)
END
```

```
TO SFUN? :DEFINITION
  OP EQUALP FIRST :DEFINITION "SFUN
END
```

```
TO NARGS :DEFINITION
  OP FIRST BF :DEFINITION
END
```

```
TO SFUN.FUNC :DEFINITION
  OUTPUT FIRST BF BF :DEFINITION
END
```

```
TO UFUN.ARGLIST :DEFINITION
  OUTPUT FIRST BF BF :DEFINITION
END
```

```
TO UFUN.BODY :DEFINITION
  OUTPUT FIRST BF BF BF :DEFINITION
END
```

```
TO APPLY :DEFINITION :VALUES
  IF SFUN? :DEFINITION [OP APPLY.SFUN ►
  :DEFINITION :VALUES]
  OP APPLY.UFUN :DEFINITION :VALUES
END
```

```
TO APPLY.SFUN :DEF :VALUES
  OUTPUT RUN SE SFUN.FUNC :DEF ( QUOTIFY ►
  :VALUES )
END
```

```
TO QUOTIFY :VALS
  IF EMPTY? :VALS [OUTPUT []]
  OUTPUT FPUT QUOTIFY1 FIRST :VALS ►
  QUOTIFY BF :VALS
END
```

```
TO QUOTIFY1 :VAL
  IF NUMBERP :VAL [OP :VAL]
  IF WORDP :VAL [OP WORD " " :VAL]
  OUTPUT :VAL
END
```

```
TO EVAL.ARGS :NARGS
  IF EQUALP :NARGS 0 [OP []]
  OUTPUT FPUT EVAL.NEXT.ITEM EVAL.ARGS ►
  :NARGS - 1
END
```

```
TO APPLY.UFUN :DEF :VALUES
  BIND.ARGS UFUN.ARGLIST :DEF :VALUES
  OP CLEANUP EVAL.BODY UFUN.BODY :DEF
END
```



```

TO BIND.ARGS :ARGLIST :VALUES
  PUSH "BIND.STACK BIND.FRAME :ARGLIST
  SET.ARGS :ARGLIST :VALUES
END

```

```

TO SET.ARGS :NAMES :VALUES
  IF EMPTY? :NAMES [STOP]
  SET FIRST :NAMES FIRST :VALUES
  SET.ARGS BF :NAMES BF :VALUES
END

```

```

TO CLEANUP :VALUE
  UNBIND
  OP :VALUE
END

```

```

TO BIND.FRAME :ARGLIST
  IF EMPTY? :ARGLIST [OP []]
  OP FPUT BIND.ARG FIRST :ARGLIST ►
    BIND.FRAME BF :ARGLIST
END

```

```

TO BIND.ARG :NAME
  OP LIST :NAME GET.VARIABLE.VALUE :NAME
END

```

```

TO UNBIND
  UNBIND.ARGS POP "BIND.STACK
END

```

```

TO UNBIND.ARGS :FRAME
  IF EMPTY? :FRAME [STOP]
  UNBIND.ARG FIRST :FRAME
  UNBIND.ARGS BF :FRAME
END

```

```

TO UNBIND.ARG :PAIR
  SET FIRST :PAIR FIRST BF :PAIR
END

```

```

TO EVAL.BODY :LINES
  OP EVAL.BODY1 "$NOVALUE :LINES
END

```

```

TO EVAL.BODY1 :VALUE :LINES
  IF EMPTY? :LINES [OP :VALUE]
  OP EVAL.BODY1 EVLINE FIRST :LINES BF ►
    :LINES
END

```

```

TO %PRINT :ARG
  PRINT :ARG
  OUTPUT :ARG
END

```

```

TO %IF :PRED :C1 :C2
  IF :PRED [OP EVLINE :C1] [OP EVLINE ►
    :C2]
END

```

```

TO %MAKE :SYM :VAL
  SET :SYM :VAL
  OUTPUT :VAL
END

```

```

TO %FD :N
  FD :N OP "TRUE
END

```

```

TO %RT :N
  RT :N OP "TRUE
END

```

```

TO %CS
  CS OP "TRUE
END

```

```

TO %TO
  OP TO1 NEXT.ITEM GATHER.ARGS
END

```

```

TO TO1 :NAME :ARGLIST
  DEF.UFUN :NAME :ARGLIST READ.BODY
  PRINT (SE :NAME "DEFINED)
  OP "TRUE
END

```

```

TO GATHER.ARGS
  IF EMPTY? :LINE [OP []]
  OP FPUT UNDOT NEXT.ITEM GATHER.ARGS
END

```

```

TO DEF.SFUN :NAME :NARGS :FUNC
  FSET :NAME MAKE.SFUN.DEF :NARGS :FUNC
END

```

```

TO DEF.UFUN :NAME :ARGS :BODY
  FSET :NAME MAKE.UFUN.DEF :ARGS :BODY
END

```

```

TO INIT
  INITPRIMS
END

```

```

TO DEF.SSFUN :NAME :NARGS
  DEF.SFUN :NAME :NARGS :NAME
END

```

```

TO INITPRIMS
DEF.SSFUN "SUM 2
DEF.SSFUN "PRODUCT 2
DEF.SSFUN "EMPTY 1
DEF.SSFUN "EQUALP 2
DEF.SSFUN "LIST 2
DEF.SSFUN "FIRST 1
DEF.SSFUN "BF 1
DEF.SSFUN "SE 2
DEF.SSFUN "WORD 2
DEF.SFUN "RT 1 "%RT
DEF.SFUN "FD 1 "%FD
DEF.SFUN "CS 0 "%CS
DEF.SFUN "THING 1 "GET.VARIABLE.VALUE
DEF.SFUN "MAKE 2 "%MAKE
DEF.SFUN "PRINT 1 "%PRINT
DEF.SFUN "IF 3 "%IF
DEF.SFUN "TO 0 "%TO
END

TO READ.BODY
OP READ.BODY1 READ.LINE
END

TO READ.BODY1 :LINE
IF EQUALP :LINE [END] [OP []]
OP FPUT :LINE READ.BODY1 READ.LINE
END

TO GET.LINE
OP INPUT.LINE "?
END

TO READ.LINE
OP INPUT.LINE ">
END

TO INPUT.LINE :PROMPT
TYPE :PROMPT
OP INPUT.LINE1 RL
END

TO INPUT.LINE1 :INPUT
IF NOT EMPTY :INPUT [OP :INPUT]
OP INPUT.LINE :PROMPT
END

TO PUSH :STACK :ITEM
MAKE :STACK FPUT :ITEM THING :STACK
END

TO POP :STACK
OP POP1 :STACK THING :STACK
END

TO POP1 :STACK :LIST
MAKE :STACK BF :LIST
OP FIRST :LIST
END

TO IGNORE :X
END

```

Map

Have you ever written a procedure like this:

```

TO LINEPRINT :LIST
IF EMPTY :LIST [STOP]
PRINT FIRST :LIST
LINEPRINT BF :LIST
END

```

By Brian Harvey.

Or like this:

```
TO TUNE :NOTES
IF EMPTY :NOTES [STOP]
TOOT 0 FIRST :NOTES 15 30
TUNE BF :NOTES
END
```

Or like this:

```
TO FLASH :COLORS
IF EMPTY :COLORS [STOP]
WAIT 60
SETBG FIRST :COLORS
FLASH BF :COLORS
END
```

All of these procedures have a common pattern. They go through a list, doing something with each member of the list, and then stop when they get to the end of the list. The procedures *differ* in what they do with the members of their input list. In one case it's a list of things to print; in the second it's a list of frequencies of musical notes; in the third it's a list of color numbers. But they all share this structure:

```
TO procedure.name :LIST
IF EMPTY :LIST [STOP]
do.something.with FIRST :LIST
procedure.name BF :LIST
END
```

You can think of this skeleton procedure as a template for many procedures that do similar work for you.

Mapping Commands

You can write a single procedure that does all these things. What's special about it is that it is a *general* tool that can apply *any* procedure to each member of a list. This general process is called *mapping* the procedure over the list, so we call this general procedure MAP. Here are some examples.

```
?MAP [PRINT] [VANILLA CHOCOLATE GINGER LEMON]
VANILLA
CHOCOLATE
GINGER
LEMON
?
```

```
?MAP [PRINT FIRST] [VANILLA CHOCOLATE GINGER LEMON]
V
C
G
L
?
```


PROGRAMMING IDEAS

```
?MAP [TYPE FIRST] [EVERY GOOD BOY DOES FINE]
EGBDF?
```

The first example of using MAP is equivalent to the procedure LINEPRINT with which we started this discussion. The first input to MAP says what you want to do to each member of the input list (in this example, PRINT it). The second input is the list over which you are mapping. So the instruction

```
MAP [PRINT] [THIS IS A LIST]
```

is equivalent to

```
LINEPRINT [THIS IS A LIST]
```

Here are the procedure definitions.

```
TO MAP :TEMPLATE :LIST
IF EMPTY? :LIST [STOP]
RUN LPUT QUOTED FIRST :LIST :TEMPLATE
MAP :TEMPLATE BF :LIST
END
```

```
TO QUOTED :THING
IF LISTP :THING [OP :THING]
OP WORD " " :THING
END
```

You can use MAP with more complicated instructions than just PRINT. In the second example, the first input to MAP is the list [PRINT FIRST]. This example works as if we'd written a special procedure like this:

```
TO FIRST.PRINT :LIST
IF EMPTY? :LIST [STOP]
PRINT FIRST FIRST :LIST
FIRST.PRINT BF :LIST
END
```

We can use MAP to obtain the same effect as the FLASH procedure we showed earlier.

```
MAP [WAIT 60 SETBG] [0 88 74 7]
```

To get the same effect as our TUNE procedure, we have to work a little harder. The problem is that the frequency input to T00T comes in the middle of the instruction, like this:

```
T00T 0 FIRST :NOTES 15 30
```

MAP expects to put each member of the list at the end of an instruction, not in the middle. What we have to do is write an auxiliary procedure that takes the frequency as a single input:

```
TO NOTE :FREQ
T00T 0 :FREQ 15 30
END
```

Now we can use MAP to get the same effect as TUNE:

```
MAP [NOTE] [440 880 220 440]
```

How It Works

What makes it possible for MAP to be a general-purpose tool instead of a procedure for a specific purpose is its use of Logo's RUN command. This replaces the specific commands like PRINT or TOUT or SETBG in the earlier examples. The input to RUN is a Logo instruction that is assembled out of two parts: the *template*, which is the first input to MAP, and one member of the list, which is MAP's second input.

Let's look at an example. If we say

```
MAP [PRINT] [VANILLA CHOCOLATE GINGER LEMON]
```

then MAP has to carry out these four instructions:

```
PRINT "VANILLA
PRINT "CHOCOLATE
PRINT "GINGER
PRINT "LEMON
```

Each of these four instructions is made by combining the template [PRINT] with one member of [VANILLA CHOCOLATE GINGER LEMON]. The combination is made using LPUT, which adds the list member at the end of the template. For example, the expression

```
LPUT ""VANILLA [PRINT]
```

outputs the list

```
[PRINT "VANILLA]
```

The procedure MAP itself has much the same pattern as the examples at the beginning of this discussion. The first instruction inside MAP is the IF EMPTY? stop rule; the last instruction is the recursive use of MAP with the BUTFIRST of the input list. Compare MAP with LINEPRINT, for example:

```
TO LINEPRINT :LIST
IF EMPTY? :LIST [STOP]
PRINT FIRST :LIST
LINEPRINT BF :LIST
END
```

```
TO MAP :TEMPLATE :LIST
IF EMPTY? :LIST [STOP]
RUN LPUT QUOTED FIRST :LIST :TEMPLATE
MAP :TEMPLATE BF :LIST
END
```

PROGRAMMING IDEAS

One possibly confusing detail in MAP has to do with quotation marks. Notice that if you want Logo to print the word VANILLA, you can't say

```
PRINT VANILLA
```

Wrong!

but must quote the input to PRINT:

```
PRINT "VANILLA
```

To assemble this instruction, the first input to LPUT must be the word "VANILLA, including the quotation mark as part of the word. The procedure QUOTED is used by MAP to supply the needed quotation marks.

Mapping Operations

So far, the templates we've used have been *commands*. That is, they have been Logo procedures that do something external, like print something, make a sound, or change the color of the screen. An even more powerful facility is to map *operations* over a list, producing (outputting) a new list of the results. Perhaps an example will make this clearer.

```
?SHOW MAP.LIST [FIRST] [THIS IS A LIST]
[T I A L]
?
```

```
?SHOW MAP.LIST [SQRT] [1 2 3 4]
[1 1.414214 1.732051 2]
?
```

Like MAP, MAP.LIST generalizes a common pattern of Logo procedures. The examples here could have been written as special-purpose procedures this way:

```
TO EVERY.FIRST :LIST
IF EMPTY? :LIST [OP []]
OP FPUT (FIRST FIRST :LIST) (EVERY.FIRST BF :LIST)
END
```

```
TO EVERY.SQRT :LIST
IF EMPTY? :LIST [OP []]
OP FPUT (SQRT FIRST :LIST) (EVERY.SQRT BF :LIST)
END
```

MAP.LIST is an operation. Its output is a list of the same length as its second input. Each member of the output list is the result of applying the template to a member of the input list.

MAP.LIST itself follows the same pattern it generalizes.

```
TO MAP.LIST :TEMPLATE :LIST
IF EMPTY? :LIST [OP []]
OP FPUT (RUN LPUT QUOTED FIRST :LIST :TEMPLATE)
(MAP.LIST :TEMPLATE BF :LIST)
END
```

Here the first input to FPUT is the same expression that was used to assemble the instructions in MAP.

An example of using MAP.LIST to apply a procedure to each word of a sentence is this program to translate a sentence into Pig Latin.

```
TO PIGLATIN :WORD
IF MEMBERP FIRST :WORD [A E I O U Y] [OP WORD :WORD "AY]
OP PIGLATIN WORD BF :WORD FIRST :WORD
END

?PRINT PIGLATIN 'HELLO
ELLOHAY
?PRINT MAP.LIST [PIGLATIN] [THIS IS GREEK TO ME]
ISTHAY ISAY EEKGRAY OTAY EMAY
?
```

Mapping Over Words

In Logo, we can assemble letters into words, just as we can assemble words into lists. We can extend the idea of mapping to apply a procedure to each letter of a word.

```
TO MAP.WORD :TEMPLATE :WORD
IF EMPTY? :WORD [OP ""]
OP WORD (RUN LPUT QUOTED FIRST :WORD :TEMPLATE)
(MAP.WORD :TEMPLATE BF :WORD)
END
```

MAP.WORD is the same as MAP.LIST, except that it uses WORD instead of FPUT as the combining operation, and it builds onto an empty word instead of an empty list.

Here is an example of how to use MAP.WORD. Suppose you want to print a word in inverse video (black on white). On the Atari computer, to print any character in inverse video, you must add 128 to the code that represents that character.

```
?PRINT MAP.WORD [CHAR 128+ASCII] 'HELLO
```

```
HELLO
```

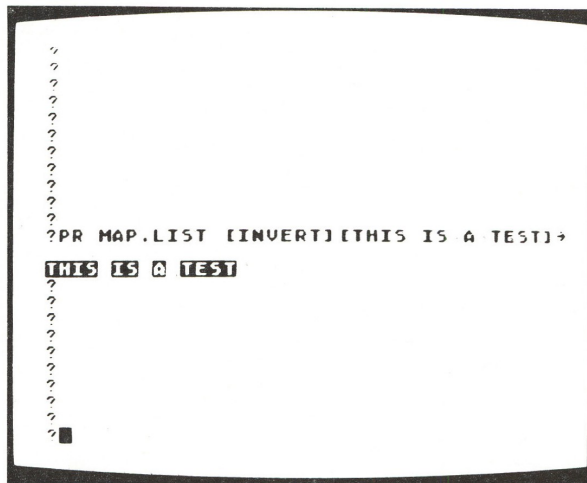
```
?
```

PROGRAMMING IDEAS

If we put this into a procedure, we can print an entire sentence with each word inverted by combining MAP.WORD and MAP.LIST.

```
TO INVERT :WORD
OP MAP.WORD [CHAR 128+ASCII] :WORD
END
```

```
?PRINT MAP.LIST [INVERT] [THIS IS A TEST.]
```

*List Reduction*

There is one more way in which an operation can be applied to the members of a list. Consider an operation with two inputs, like SUM or PRODUCT. It is often convenient to be able to add up all the numbers in a list, or multiply them together. Of course, as in the earlier situations, we could write special-purpose procedures.

```
TO ADD :LIST
IF EMPTY? :LIST [OP 0]
OP SUM (FIRST :LIST) (ADD BF :LIST)
END
```

```
TO MULTIPLY :LIST
IF EMPTY? :LIST [OP 1]
OP PRODUCT (FIRST :LIST) (MULTIPLY BF :LIST)
```

```
?PR ADD [1 2 3 4]
10
?PR MULTIPLY [1 2 3 4]
24
?
```

What we'd like to do is produce a general tool for these situations.

```
?PR REDUCE [SUM] [1 2 3 4]
10
```

```
?PR REDUCE [PRODUCT] [1 2 3 4]
24
?
```

There is one slight complication that prevents REDUCE from following exactly the pattern of ADD and MULTIPLY. The problem is that each of those procedures knows about the *identity element* for the corresponding operation. The identity element is the value to start with when the input list is empty: 0 for SUM, 1 for PRODUCT. To make REDUCE a general tool, we want to avoid building this kind of information into it. The solution is to apply REDUCE recursively only down to the point where there are *two* members remaining in the input list, then just apply the template to those two. The resulting procedure is a little messy, but if you go through it carefully you'll see that it's really much like the mapping procedures we've used before.

```
TO REDUCE :TEMPLATE :LIST
IF EMPTY? BF :LIST [OP FIRST :LIST]
IF EMPTY? BF BF :LIST [OP RUN SE :TEMPLATE
LIST (QUOTED FIRST :LIST) (QUOTED FIRST BF :LIST)]
OP RUN SE :TEMPLATE LIST (QUOTED FIRST :LIST)
(QUOTED REDUCE :TEMPLATE BF :LIST)
END
```

Here are more examples of how REDUCE can be used.

```
?PRINT REDUCE [WORD] [A B C D]
ABCD
?
```

```
TO REVERSE :LIST
OP REDUCE [LPUT] LPUT [] :LIST
END
```

```
?SHOW REVERSE [A B C D]
[D C B A]
?
```

SUGGESTIONS

- You could modify these procedures so that the list members could be inserted anywhere in the template, instead of only at the end. For example, the music example that earlier required writing an auxiliary procedure NOTE could instead be written

```
MAP [T00T 0 ? 15 30] [440 880 220 440]
```

where the question mark indicates the position in the template into which the members of the input list are placed.

- The general name for doing something over and over is *iteration*. Mapping is a particular kind of iteration, based on using the members of a list, one after the other. Other kinds of iteration can also be

invented using the RUN primitive. For example, here is an iteration procedure that tests a predicate to control the repetition.

```
TO WHILE :PREDICATE :COMMAND
  IF NOT RUN :PREDICATE [STOP]
  RUN :COMMAND
  WHILE :PREDICATE :COMMAND
END

?CS
?WHILE [HEADING < 270] [FD 10 RT 10]
?
```

You might try to write a procedure to create numeric iteration.

```
?STEP "NUM 3 7 [PRINT :NUM * :NUM]
9
16
25
36
49
?
```

- Use MAP.WORD and MAP.LIST to implement a *substitution cipher*. A cipher is a technique for protecting secret messages by transforming each letter into some other form. (Ciphers are sometimes called *codes*, but, strictly speaking, a code is a technique that transforms a word by looking it up in a dictionary, rather than by manipulating it letter by letter. A foreign language is like a code.) Write a procedure that takes a single letter as input and outputs some secret representation of the input letter. Then you can encipher a word by applying MAP.WORD to it, and you can encipher a sentence by applying MAP.LIST to encipher each word. The example of inverse video works like a cipher, although of course the result isn't very secret.
- MAP.LIST uses FPUT to accumulate the results for each member of the input list, and MAP.WORD uses WORD to accumulate its results. Logo has other accumulating operations: SE, LIST, and LPUT. Try writing versions of MAP.LIST that use each of these. Are any of them useful?
- Here is a tricky example.

```
TO FLATTEN :LIST
  IF WORDP :LIST [OP :LIST]
  OP REDUCE [SE] MAP.LIST [FLATTEN] :LIST
END

?SHOW FLATTEN [[THIS IS] [A [LIST]]]
[THIS IS A LIST]
?
```

FLATTEN combines iteration over a list, list reduction, and recursion, since the template input to MAP.LIST uses FLATTEN itself. The procedure converts any list into a *flat list*, one that has only words as

members. Can you see why both REDUCE and MAP.LIST must be used? Compare the result of FLATTEN to these:

```
SHOW REDUCE [SE] [[THIS IS] [A [LIST]]]
SHOW MAP.LIST [FLATTEN] [[THIS IS] [A [LIST]]]
```

PROGRAM LISTING

<pre>TO MAP :TEMPLATE :LIST IF EMPTY? :LIST [STOP] RUN LPUT QUOTED FIRST :LIST :TEMPLATE MAP :TEMPLATE BF :LIST END</pre>	<pre>TO MAP.WORD :TEMPLATE :WORD IF EMPTY? :WORD [OP ""] OP WORD (RUN LPUT QUOTED FIRST :WORD ► :TEMPLATE) (MAP.WORD :TEMPLATE BF ► :WORD) END</pre>
<pre>TO QUOTED :THING IF LISTP :THING [OP :THING] OP WORD "" :THING END</pre>	<pre>TO REDUCE :TEMPLATE :LIST IF EMPTY? BF :LIST [OP FIRST :LIST] IF EMPTY? BF BF :LIST [OP RUN SE ► :TEMPLATE LIST (QUOTED FIRST ► :LIST) (QUOTED FIRST BF :LIST)] OP RUN SE :TEMPLATE LIST (QUOTED FIRST ► :LIST) (QUOTED REDUCE :TEMPLATE ► BF :LIST) END</pre>
<pre>TO MAP.LIST :TEMPLATE :LIST IF EMPTY? :LIST [OP []] OP FPUT (RUN LPUT QUOTED FIRST :LIST ► :TEMPLATE) (MAP.LIST :TEMPLATE BF ► :LIST) END</pre>	

Mergesort

People often want to use computers to *sort* information of various kinds. For example, you may want to list your friends' addresses in alphabetical order, or you may want the same information arranged in order of their birthdays to remind you when to send cards. Programmers have invented many different techniques to solve the sorting problem. Generally, the methods that are easy to understand tend to run slowly, while the faster methods are rather complicated. Here is a method that is medium-fast and medium-tricky. Its name is *mergesort*.

In Logo, we'll represent the information we want to sort as a list of items. The general strategy is this:

1. Divide the list into two smaller parts.
2. Sort each part separately.
3. Merge the two sorted lists into one big sorted list.

This may not seem like much of a strategy, because we are still left with the problem of sorting the smaller lists in the second step. But the clever part

PROGRAMMING IDEAS

is that if we keep applying the strategy to the smaller lists, eventually we get lists with just one member, and we can simply declare these lists sorted.

Here's a specific example. To make it easy to read, we'll sort a list of numbers in size order. Start with this list:

```
[14 3 27 1 10 5]
```

Divide it into two smaller lists.

```
[14 3 27]           [1 10 5]
```

Now sort the first of the smaller lists. To do that, divide it into two smaller lists.

```
[14 3]              [27]
```

Now sort the first of *these* lists, again by dividing it into two smaller lists.

```
[14]                [3]
```

Each of these lists has only one member, so each is already sorted. Now we merge them to get

```
[3 14]
```

Now we can merge this list with its "partner," which is the list [27]. The result is

```
[3 14 27]
```

The next step is to sort the "partner" of *this* list, namely the list [1 10 5]. This also involves dividing it into smaller lists, as before. To make this example shorter, we'll skip the steps of sorting the list [1 10 5]. Finally we are left with two sorted lists:

```
[3 14 27]           [1 5 10]
```

The last step is to merge these:

```
[1 3 5 10 14 27]
```

Dividing a List into Two Parts

The first step in the sorting process is to divide a list into two parts. To do that, we can use procedures FIRST.PART and LAST.PART.

```
TO FIRST.PART :LIST
OP FIRST.N (INT (COUNT :LIST)/2) :LIST
END
```

```
TO FIRST.N :NUMBER :LIST
IF :NUMBER=0 [OP :LIST]
OP FIRST.N :NUMBER-1 BL :LIST
END
```

```
TO LAST.PART :LIST
OP LAST.N (INT (1+COUNT :LIST)/2) :LIST
END
```



```

TO LAST.N :NUMBER :LIST
IF :NUMBER=0 [OP :LIST]
OP LAST.N :NUMBER-1 BF :LIST
END

```

You may notice that LAST.PART refers to 1 + COUNT :LIST instead of COUNT :LIST. The reason for this difference is that if the input list has an odd number of members, we must divide the list into two pieces that differ in length by one. For example, if the input list has five members, FIRST.PART will output the first three members of the list and LAST.PART will output the last two members.

```

?SHOW FIRST.PART [14 3 27 1 10]
[14 3 27]
?SHOW LAST.PART [14 3 27 1 10]
[1 10]
?

```

Merging Two Ordered Lists

The last step of the sorting procedure is to *merge* two lists. The MERGE procedure assumes that each of the two lists is already in the correct order. MERGE takes two inputs, namely, the two lists.

MERGE compares the first member of one input list with the first member of the other list. One of these becomes the first member of the final merged list; MERGE is applied recursively to the remaining members of the input lists.

```

TO MERGE :A :B
IF EMPTY? :A [OP :B]
IF EMPTY? :B [OP :A]
IF COMPARE FIRST :A FIRST :B
  [OP FPUT FIRST :A MERGE BF :A :B]
OP FPUT FIRST :B MERGE :A BF :B
END

```

MERGE uses a subprocedure, COMPARE, which tells whether one item should come before or after another. COMPARE takes two inputs. It outputs the word TRUE if the first input comes before the second input, or FALSE otherwise.

You can write different versions of COMPARE depending on what ordering you want to use for your sorted lists. If you are sorting numbers by size, as in the earlier example, you can use this version:

```

TO COMPARE :A :B
OUTPUT :A < :B
END

```

If you want to sort words alphabetically, or use some other ordering, you need a more complicated version of COMPARE. We'll show an example later.

PROGRAMMING IDEAS

Putting It All Together

We've written the easy parts of this sorting method. The hard part is putting it all together. The main procedure SORT does this. It takes one input, which must be a list. It outputs the same list, but with its members in sorted order.

```
TO SORT :A
  IF EMPTY? :A [OP []]
  IF EMPTY? BF :A [OP :A]
  OP MERGE (SORT FIRST.PART :A) (SORT LAST.PART :A)
END
```

If the input list is empty, or has only one member, then the list is already sorted. SORT outputs the list unchanged. For larger lists, SORT goes through the steps we described at the beginning.

1. It uses FIRST.PART and LAST.PART to divide the list in two.
2. It uses SORT to sort each of these smaller lists.
3. It uses MERGE to combine the resulting ordered lists.

Alphabetical Order

Sometimes we want to deal with information composed of words or sentences, rather than numbers. Here are procedures to alphabetize lists of words.

```
TO COLLATE.BEFORE :A :B
  IF EMPTY? :A [OP "TRUE]
  IF EMPTY? :B [OP "FALSE]
  IF COLLATE.BEFORE.WORD FIRST :A FIRST :B [OP "TRUE]
  IF NOT EQUALP FIRST :A FIRST :B [OP "FALSE]
  OP COLLATE.BEFORE BF :A BF :B
END
```

```
TO COLLATE.BEFORE.WORD :A :B
  IF EMPTY? :A [OP "TRUE]
  IF EMPTY? :B [OP "FALSE]
  IF (ASCII FIRST :A) < (ASCII FIRST :B) [OP "TRUE]
  IF NOT EQUALP FIRST :A FIRST :B [OP "FALSE]
  OP COLLATE.BEFORE.WORD BF :A BF :B
END
```

COLLATE.BEFORE takes two inputs. Each input is a sentence (in other words, a list of words). It outputs the word TRUE if the first input comes before the second alphabetically.

COLLATE.BEFORE.WORD is similar to COLLATE.BEFORE, except that its two inputs are single words instead of lists of words.

An Example

Here is a list of the greatest songs of all time.


```
MAKE "RECORDS [
  [[SHE LOVES YOU] [BEATLES]]
  [[SHE'S NOT THERE] [ZOMBIES]]
  [[WATERLOO SUNSET] [KINKS]]
  [[FLYING ON THE GROUND IS WRONG]
   [BUFFALO SPRINGFIELD]]
  [[MY GENERATION] [WHO]] ]
```

(To type in a long list like this, you have to use the Logo editor. When you are typing directly to the ? prompt in Atari Logo, there is a limit to how long a line you can type.)

This list contains five items. Each item is itself a list with two members, the title and artist of a record. This is a simple example of a *data structure*. That is, instead of having a list of words or a list of numbers, we have a list of more complicated things, each of which is itself made up of smaller parts.

Suppose we want to sort these songs by title. We can define a COMPARE procedure to do that.

```
TO COMPARE :A :B
OP COLLATE.BEFORE FIRST :A FIRST :B
END
```

The FIRST of each song is a list containing its title, so this version of COMPARE sees which title comes first alphabetically.

```
?SHOW SORT :RECORDS
[[[FLYING ON THE GROUND IS WRONG] [BU->
FFALO SPRINGFIELD]] [[MY GENERATION] ->
[WHO]] [[SHE LOVES YOU] [BEATLES]] [[->
SHE'S NOT THERE] [ZOMBIES]] [[WATERLO->
O SUNSET] [KINKS]]]
?
```

We can make this prettier by using a formatting procedure to print each record on a separate line.

```
TO FORMAT :LIST
IF EMPTY? :LIST [STOP]
PRINT SE "TITLE: FIRST FIRST :LIST
PRINT SE "...ARTIST: LAST FIRST :LIST
FORMAT BF :LIST
END
```

```
?FORMAT SORT :RECORDS
TITLE: FLYING ON THE GROUND IS WRONG
...ARTIST: BUFFALO SPRINGFIELD
TITLE: MY GENERATION
...ARTIST: WHO
TITLE: SHE LOVES YOU
...ARTIST: BEATLES
TITLE: SHE'S NOT THERE
...ARTIST: ZOMBIES
TITLE: WATERLOO SUNSET
...ARTIST: KINKS
?
```


PROGRAMMING IDEAS

Now suppose we want to sort the same list of records, this time by artist. To do this, we replace the COMPARE procedure with one that uses the LAST of each item instead of the FIRST.

```
TO COMPARE :A :B
OP COLLATE.BEFORE LAST :A LAST :B
END
```

The LAST of each song is a list containing the name of the group that performed it.

```
?FORMAT SORT :RECORDS
TITLE: SHE LOVES YOU
...ARTIST: BEATLES
TITLE: FLYING ON THE GROUND IS WRONG
...ARTIST: BUFFALO SPRINGFIELD
TITLE: WATERLOO SUNSET
...ARTIST: KINKS
TITLE: MY GENERATION
...ARTIST: WHO
TITLE: SHE'S NOT THERE
...ARTIST: ZOMBIES
?
```

SUGGESTIONS

If you are interested in learning about other ways to write sorting programs, the standard reference book on this subject is *Sorting and Searching*, volume 3 of *The Art of Computer Programming*, by Donald E. Knuth (Reading, Mass.: Addison-Wesley, 1973).

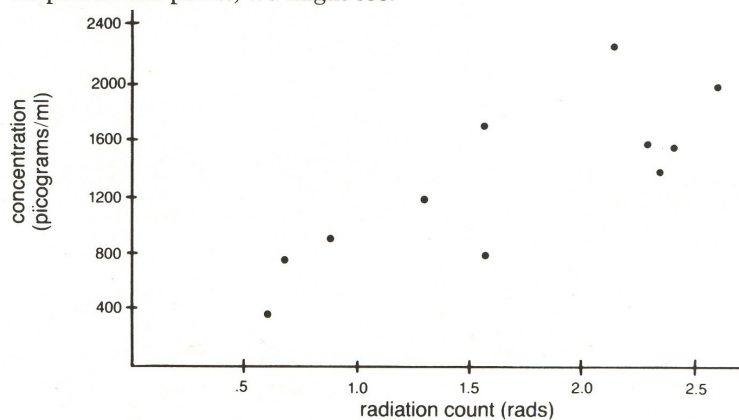
PROGRAM LISTING

Note: There are three different versions of COMPARE in the write-up. The one here is the first version. COMPARE2 and COMPARE3 are the other two versions and can be substituted for COMPARE in MERGE.

TO FIRST.PART :LIST	TO LAST.N :NUMBER :LIST
OP FIRST.N (INT (COUNT :LIST)/2) :LIST	IF :NUMBER=0 [OP :LIST]
END	OP LAST.N :NUMBER-1 BF :LIST
	END
TO FIRST.N :NUMBER :LIST	TO MERGE :A :B
IF :NUMBER=0 [OP :LIST]	IF EMPTY :A [OP :B]
OP FIRST.N :NUMBER-1 BL :LIST	IF EMPTY :B [OP :A]
END	IF COMPARE FIRST :A FIRST :B [OP FPUT ►
TO LAST.PART :LIST	FIRST :A MERGE BF :A :B]
OP LAST.N (INT (1+COUNT :LIST)/2) ►	OP FPUT FIRST :B MERGE :A BF :B
:LIST	END
END	

Making a Graph of the Data

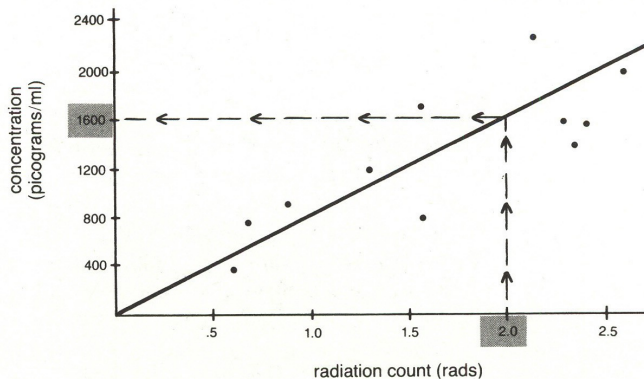
In the antibody experiment, we take samples of known antibody concentrations, measure their radiation counts, and plot them on a graph. For each known concentration, we plot the corresponding radiation count.* When we plot all the points, we might see:



We can use this plot for looking at the data from our samples of known concentrations. How can we use this data to estimate the *unknown* concentrations of our experimental samples?

We know that for this kind of experiment, the radiation count of a sample is proportional to the concentration of antibodies in it. That is, when we double the concentration, the radiation emitted will be doubled. This relationship suggests that the graph of radiation versus concentration is a *line*. We need to find a line on which we can look up an estimate of the concentration of a sample once we have experimentally found its radiation count.

Looking Things Up on a Graph



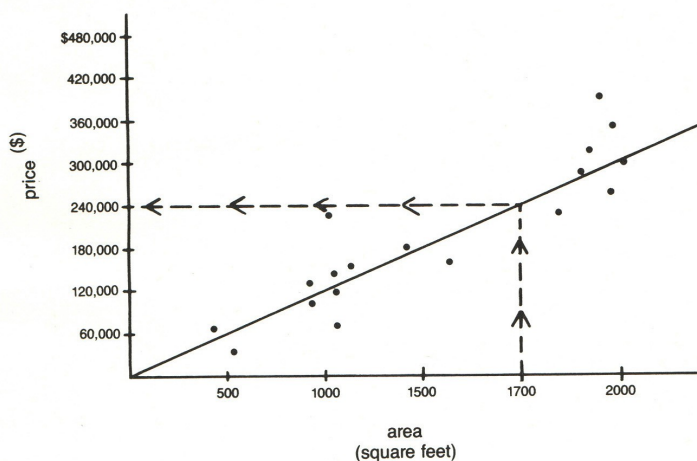
Let's look at the graph above. It shows a regression line plotted for the data points in this experiment.* Once we know how much radiation is emitted,

*The "best-fitting" line through a sample of data points is called a "least squares," or regression, line and is calculated from the data points.

we can estimate the concentration of antibodies present. For example, if the radiation count is 2, then the concentration is estimated to be 1600 picograms/ml.

This line was already calculated for this particular experiment; someone plotted the data points and determined the line. Different samples of data generate different graphs and regression lines.

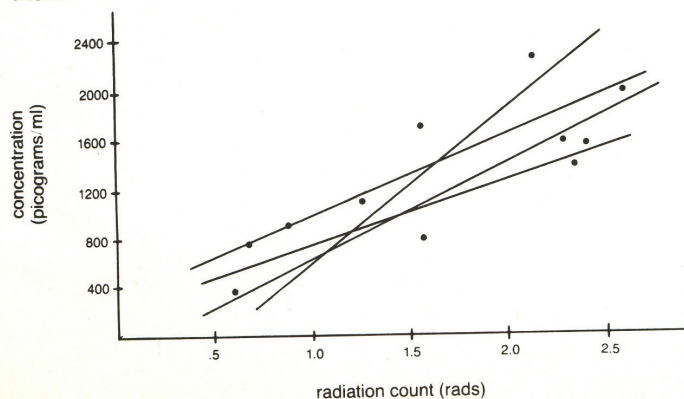
For example, a realtor selling office space might want to know how much to charge for a 1700-square-foot building. Let's say the realtor called other realtors who sold office space in the same community and asked them how much they charged for buildings of different square footages. A helpful graph would be price plotted against square footage. While the realtor might consider other factors in setting the price (for example, property location, condition of the building), she is able to estimate the market price for the office space.



Possible Lines

Many different lines might be drawn through the known sample points. How can we find a line that goes through all the measured points and makes sense for our data?

Since this is a real-world experiment, the sample points don't all lie exactly on a straight line. We could draw lots of lines near these data points. We would like to find the one line that goes as close as possible to *all* of them.



Moving the line closer to some points will increase its distance from others. Some of the lines fit the data so poorly that we wouldn't even consider them. Others would seem to be pretty good fits to the data. We need to find a way of determining the line with the best fit, the line that comes closest to all the points. How can we choose which line is the best fit?

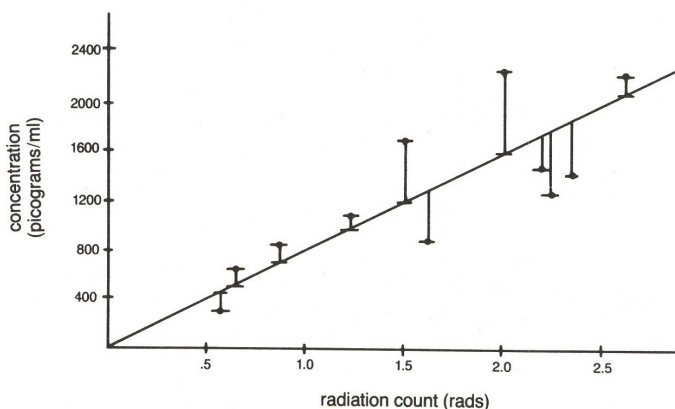
A Technique for Finding the Best Line

There are two things you need to know to plot a line: its slope (m) and its y -intercept (b). The standard equation for a line is

$$y = mx + b$$

Once you know m and b , you can use the equation to find the y coordinate for any x .

A method usually used to find the best-fit line is called "least squares." The least squares line is that which minimizes the sum of the squares of the vertical distances between the line and the data points. It is a neat way of solving the problem when the real-world data points are not exactly on the line (this is usually called minimizing the error). Let's look at the following graph.



Not all the data points fall on the regression line. The amount of "error" of the regression line is the sum of the squares of the vertical distance of each point from the line. If all the data points fall on the best-fit line the error would be 0. This is the ideal; most real-world data do not behave so neatly. The method of least squares is used to calculate a line that *minimizes* the error.

Given a set of points, you can find the best-fit least squares line by solving two equations: one to calculate the slope of the best-fit line and one to calculate its y -intercept.

In our experiment, we have the x and y coordinates of N points. We can use the coordinates and these two equations to find m and b :

$$m = \frac{N\sum xy - \sum x}{N\sum x^2 - (\sum x)^2}$$

$$b = \frac{\sum y - m\sum x}{N}$$

The Greek letter sigma, Σ , is called summation notation; it means that you add a set of numbers. Σx means add up all the x coordinates from the set of points. Σxy means you should multiply the x and y coordinates of each point and add up all the products.

Using the Program

Here is an example of how to use BESTLINE. The text that is boldface is what you type. Say your points are (28, 39), (25, 10), (140, 72), and (5, 2).

BESTLINE

PLEASE TYPE YOUR POINTS: X Y X Y ...

28 39 25 10 140 72 5 2

Before plotting the line, BESTLINE prints:

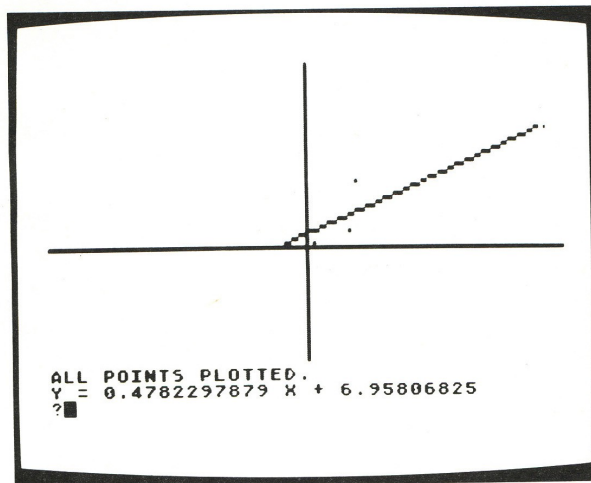
SLOPE = 0.4756966082

Y-INTERCEPT = 7.203018

$Y = 0.4756966882 X + 7.203018$

PRESS ANY KEY TO CONTINUE

When BESTLINE continues, it plots your points and draws the line that best fits them.



At the bottom of the screen BESTLINE prints:

ALL POINTS PLOTTED.

THE X FOR 2 = -10.93768151

THE X FOR 72 = 136.214933

PROGRAMMING IDEAS

To find the y coordinate on the best (fit line for a certain x -100, for example) type:

```
SOLVE.Y 100
THE Y FOR 100 = 54.77267882
```

Similarly, you can use a procedure called SOLVE.X to find the x value for a certain y .

*How the Program Works***Overview**

BESTLINE is the top-level procedure. It sets up a global variable, POINTLIST, to contain the list of points the user types in. The list of x and y coordinates the user types is converted into a list of lists by PAIRUP. Each sublist contains the x and y coordinates for each point. In our example, :POINTLIST is [[28 39] [25 10] [140 72] [5 2]]. BESTLINE calls LINE.EQUATION to find the slope and the y -intercept of the line that best fits these points. BESTLINE then calls PLOTLINE to plot the points and draw the best-fit line.

```
TO BESTLINE
HT TS CT
PR [PLEASE TYPE YOUR POINTS: X Y X Y ...]
MAKE "POINTLIST PAIRUP RL
LINE.EQUATION :POINTLIST
PR [PRESS ANY KEY TO CONTINUE]
IGNORE RC
WINDOW
PLOTLINE :POINTLIST
END

TO IGNORE :THING
END
```

LINE.EQUATION creates two global variables, M and B. :M is the slope of the line and is computed by LEAST.SQUARES.SLOPE. :B is the y -intercept and is computed by YINTERCEPT.

```
TO LINE.EQUATION :POINTLIST
MAKE "M LEAST.SQUARES.SLOPE :POINTLIST
PR SE [SLOPE IS] :M
MAKE "B YINTERCEPT :POINTLIST
PR SE [Y\-INTERCEPT =] :B
PR ( SE [Y =] :M [X +] :B )
END

TO LEAST.SQUARES.SLOPE :POINTS
MAKE "SUMX BIGE :POINTS "JUSTX
OP ( ( COUNT :POINTS ) * ( BIGE :POINTS "XTIMESY )
    - ( :SUMX * BIGE :POINTS "JUSTY ) )
    / ( ( COUNT :POINTS ) * ( BIGE :POINTS "XSQUARED )
        - ( :SUMX * :SUMX ) )
END
```

```

TO YINTERCEPT :POINTS
OP ( ( ( BIGE :POINTS "JUSTY )
      - ( :M * BIGE :POINTS "JUSTX ) ) / COUNT :POINTS )
END

```

Both LEAST.SQUARES.SLOPE and YINTERCEPT rely on a collection of procedures used by BIGE.

BIGE

BIGE takes two inputs, a list of points and the name of another procedure. It sums the result of applying that procedure to each point in the list.* (This procedure is called BIGE, pronounced "big-ee," because the Greek letter Σ , used as the summation symbol, looks like an upper-case "E." For example, if you type `BIGE :POINTS "JUSTX`, JUSTX will extract just the x coordinate from each point in the list and BIGE will end up adding up just the x 's! `BIGE :POINTS "XTIMESY` adds up the products of the x and y coordinates for each point. The procedures used with BIGE in the formulas are JUSTX, JUSTY, XTIMESY, and XSQUARED.

```

TO BIGE :LIST :PROC
IF EMPTY :LIST [OP 0]
OP ( SUM ( RUN LIST :PROC FIRST :LIST )
      BIGE BF :LIST :PROC )
END

TO JUSTX :POINT
OP FIRST :POINT
END

TO JUSTY :POINT
OP FIRST BF :POINT
END

TO XTIMESY :POINT
OP ( JUSTX :POINT ) * ( JUSTY :POINT )
END

TO XSQUARED :POINT
OP ( JUSTX :POINT ) * ( JUSTX :POINT )
END

```

Graphing

After the equation of the best-fit line is determined, PLOTLINE plots your points and the line. PLOTLINE first uses PLOT.POINTS to draw your points.

*BIGE is a mapping procedure. See Brian Harvey's Map project (p.322) for more about mapping.

PROGRAMMING IDEAS

```

TO PLOTLINE :LIST
SS
PLOT.AXES
PLOT.POINTS :LIST
WAIT 60
RANGE :LIST
IF :M = 0 [PLOT.HORIZ STOP]
PU
SETPOS LIST XVALUE :MINY :MINY
PD
SETPOS LIST XVALUE :MAXY :MAXY
PRINT (SE [Y = ] :M [X + ] :B)
END

TO PLOT.AXES
MAKE "PN PN
IF :PN = 2 [SETPN 0] [SETPN PN + 1]
SETPC PN 0
PU SETPOS [-150 0]
PD SETPOS [150 0]
PU SETPOS [0 -110]
PD SETPOS [0 110]
PU SETPN :PN HOME
END

TO PLOT.POINTS :LIST
IF EMPTY? :LIST [PR [ALL POINTS PLOTTED.] STOP]
PU
SETPOS FIRST :LIST
PD FD 0
PLOT.POINTS BF :LIST
END

TO SOLVE.Y :X
PR ( SE [THE Y FOR] :X "= ( :M * :X + :B ) )
END

TO SOLVE.X :Y
IF :M = 0 [PRINT [NO SOLUTION, M=0] STOP]
PRINT (SE [THE X FOR] :Y "= XVALUE :Y)
END

TO XVALUE :Y
OP (:Y - :B) / :M
END

```

PLOTLINE then draws the best-fit line. If the slope (:M) is 0, then PLOT.HORIZ draws the line. The procedure RANGE finds the smallest and largest x and y coordinates for your set of points. SOLVE.X finds the best-fit line's x coordinate for the minimum y and maximum y computed by RANGE. These are the procedures for finding and plotting the endpoints of the best-fit line.


```
TO RANGE :PLIST
MAKE "MINX LEAST.NUM XLIST :PLIST
MAKE "MINY LEAST.NUM YLIST :PLIST
MAKE "MAXX GREATEST.NUM XLIST :PLIST
MAKE "MAXY GREATEST.NUM YLIST :PLIST
END
```

```
TO LEAST.NUM :NUMS
IF EMPTY? BF :NUMS [OP FIRST :NUMS]
IF ( FIRST :NUMS ) < ( FIRST BF :NUMS )
    [OP LEAST.NUM SE BF BF :NUMS FIRST :NUMS]
OP LEAST.NUM BF :NUMS
END
```

```
TO GREATEST.NUM :NUMS
IF EMPTY? BF :NUMS [OP FIRST :NUMS]
IF ( FIRST BF :NUMS ) > FIRST :NUMS
    [OP GREATEST.NUM BF :NUMS]
OP GREATEST.NUM SE BF BF :NUMS FIRST :NUMS
END
```

```
TO XLIST :POINTLIST
IF EMPTY? :POINTLIST [OP []]
OP FPUT JUSTX FIRST :POINTLIST XLIST BF :POINTLIST
END
```

```
TO YLIST :POINTLIST
IF EMPTY? :POINTLIST [OP []]
OP FPUT JUSTY FIRST :POINTLIST YLIST BF :POINTLIST
END
```

PLOT.HORIZ is used in the special case when the slope of the line is 0.

```
TO PLOT.HORIZ
PU SETPOS LIST :MINX :B
PD SETPOS LIST :MAXX :B
END
```

PAIRUP and PAIRS are used by BESTLINE to convert a list of coordinates typed by the user into a list of points that the program can use.

```
TO PAIRUP :LIST
IF 1 = REMAINDER COUNT :LIST 2 [MAKE "LIST BL :LIST]
OP PAIRS :LIST
END
```

```
TO PAIRS :LIST
IF EMPTY? :LIST [OP []]
OP FPUT SE FIRST :LIST FIRST BF :LIST PAIRS BF BF :LIST
END
```

PROGRAM LISTING

```

TO BESTLINE
HT TS CT
PR [PLEASE TYPE YOUR POINTS: X Y X Y ►
...]
MAKE "POINTLIST PAIRUP RL
LINE.EQUATION :POINTLIST
PR [PRESS ANY KEY TO CONTINUE]
IGNORE RC
WINDOW
PLOTLINE :POINTLIST
END

TO IGNORE :THING
END

TO LINE.EQUATION :POINTLIST
MAKE "M LEAST.SQUARES.SLOPE :POINTLIST
PR SE [SLOPE IS] :M
MAKE "B YINTERCEPT :POINTLIST
PR SE [Y\-INTERCEPT =] :B
PR ( SE [Y =] :M [X +] :B )
END

TO LEAST.SQUARES.SLOPE :POINTS
MAKE "SUMX BIGE :POINTS "JUSTX
OP ( ( COUNT :POINTS ) * ( BIGE ►
:POINTS "XTIMESY ) - ( :SUMX * ►
BIGE :POINTS "JUSTY ) ) / ( ( ►
COUNT :POINTS ) * ( BIGE :POINTS ►
"XSQUARED ) - ( :SUMX * :SUMX ) )
END

TO YINTERCEPT :POINTS
OP ( ( ( BIGE :POINTS "JUSTY ) - ( :M ►
* BIGE :POINTS "JUSTX ) ) / COUNT ►
:POINTS )
END

TO BIGE :LIST :PROC
IF EMPTY :LIST [OP 0]
OP ( SUM ( RUN LIST :PROC FIRST :LIST ►
) BIGE BF :LIST :PROC )
END

TO JUSTX :POINT
OP FIRST :POINT
END

TO JUSTY :POINT
OP FIRST BF :POINT
END

TO XTIMESY :POINT
OP ( JUSTX :POINT ) * ( JUSTY :POINT )
END

TO XSQUARED :POINT
OP ( JUSTX :POINT ) * ( JUSTX :POINT )
END

TO PLOTLINE :LIST
SS
PLOT.AXES
PLOT.POINTS :LIST
WAIT 60
RANGE :LIST
IF :M = 0 [PLOT.HORIZ STOP]
PU
SETPOS LIST XVALUE :MINY :MINY
PD
SETPOS LIST XVALUE :MAXY :MAXY
PRINT ( SE [Y = ] :M [X + ] :B )
END

TO PLOT.AXES
MAKE "PN PN
IF :PN = 2 [SETPN 0] [SETPN PN + 1]
SETPC PN 0
PU SETPOS [-150 0]
PD SETPOS [150 0]
PU SETPOS [0 -110]
PD SETPOS [0 110]
PU SETPN :PN HOME
END

TO PLOT.POINTS :LIST
IF EMPTY :LIST [PR [ALL POINTS ►
PLOTTED.] STOP]
PU
SETPOS FIRST :LIST
PD FD 0
PLOT.POINTS BF :LIST
END

TO SOLVE.Y :X
PR ( SE [THE Y FOR] :X "= ( :M * :X + ►
:B ) )
END

TO SOLVE.X :Y
IF :M = 0 [PRINT [NO SOLUTION, M=0] ►
STOP]
PRINT ( SE [THE X FOR] :Y "= XVALUE :Y )
END

```



```

TO XVALUE :Y
OP (:Y - :B) / :M
END

TO RANGE :PLIST
MAKE "MINX LEAST.NUM XLIST :PLIST
MAKE "MINY LEAST.NUM YLIST :PLIST
MAKE "MAXX GREATEST.NUM XLIST :PLIST
MAKE "MAXY GREATEST.NUM YLIST :PLIST
END

TO LEAST.NUM :NUMS
IF EMPTY BF :NUMS [OP FIRST :NUMS]
IF ( FIRST :NUMS ) < ( FIRST BF :NUMS ►
    ) [OP LEAST.NUM SE BF BF :NUMS ►
    FIRST :NUMS]
OP LEAST.NUM BF :NUMS
END

TO GREATEST.NUM :NUMS
IF EMPTY BF :NUMS [OP FIRST :NUMS]
IF ( FIRST BF :NUMS ) > FIRST :NUMS ►
    [OP GREATEST.NUM BF :NUMS]
OP GREATEST.NUM SE BF BF :NUMS FIRST ►
    :NUMS
END

TO XLIST :POINTLIST
IF EMPTY BF :POINTLIST [OP []]
OP FPUT JUSTX FIRST :POINTLIST XLIST ►
    BF :POINTLIST
END

TO YLIST :POINTLIST
IF EMPTY BF :POINTLIST [OP []]
OP FPUT JUSTY FIRST :POINTLIST YLIST ►
    BF :POINTLIST
END

TO PLOT.HORIZ
PU SETPOS LIST :MINX :B
PD SETPOS LIST :MAXX :B
END

TO PAIRUP :LIST
IF 1 = REMAINDER COUNT :LIST 2 [MAKE ►
    "LIST BL :LIST]
OP PAIRS :LIST
END

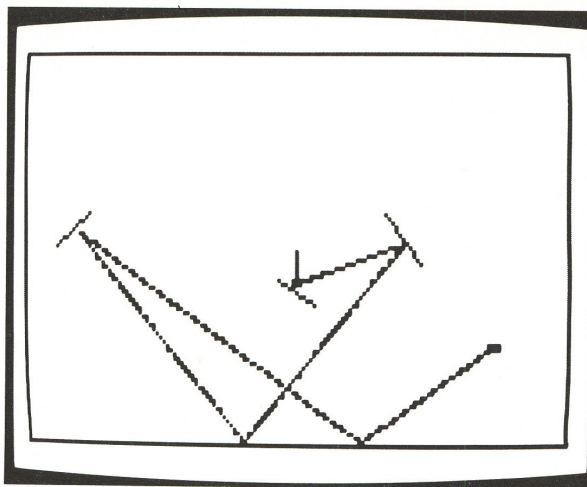
TO PAIRS :LIST
IF EMPTY BF :LIST [OP []]
OP FPUT SE FIRST :LIST FIRST BF :LIST ►
    PAIRS BF BF :LIST
END

```

Lines and Mirrors

This program was designed to simulate a beam of light bouncing off mirrors or a ball bouncing off walls. The user enters the coordinates of endpoints of lines. The program then draws the lines and starts the turtle going in a random direction. When the turtle hits one of those lines, it will bounce off at the same angle at which it came in. The turtle draws its path as it goes. You can think of the turtle's path as a beam of light and the lines as mirrors.

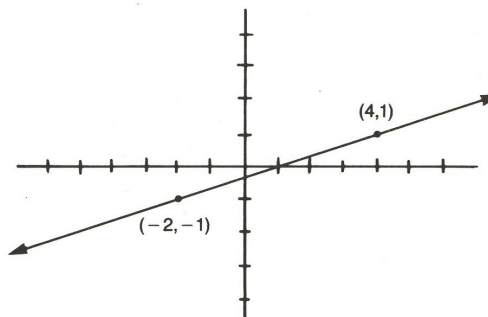
In this write-up there are three main sections: first, how the program calculates the angle at which the turtle should bounce after hitting a line; second, how the information about the lines is remembered; and third, the detailed structure of the program.



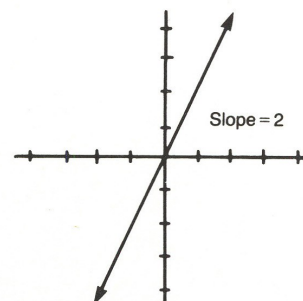
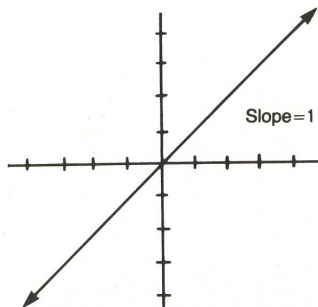
Bouncing Off a Line

What Is a Line?

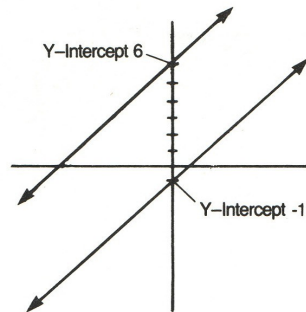
You probably know that two points determine a line, as in the following illustration.



Another way to determine a line is by its slope (m) and y -intercept (b). The slope is the steepness of the line. In the following figure the line on the right rises twice as fast as the line on the left.



There may be many different lines with the same slope. A way to distinguish these lines is by their y -intercept. The y -intercept is the point at which the line crosses the y axis.



Calculating the Turning Angle

For our purposes we need a representation that tells us *where* the line is and what its *orientation* is. (The orientation is particularly important because the problem we are trying to solve is about directions of motion.) These aspects of lines are reminiscent of the major components of the state of a turtle: position and heading. This suggests that the best way to represent the orientation of a line is by the heading that a turtle would take to draw it.

It is important for us to know the heading of a line in order to figure out how the turtle should bounce off it. The angle at which the turtle comes in (angle of incidence) should be the same as the angle at which it bounces out (angle of reflection).



The angle of incidence is the amount the turtle must turn to get from its initial heading to the heading of the line.



PROGRAMMING IDEAS

So we get

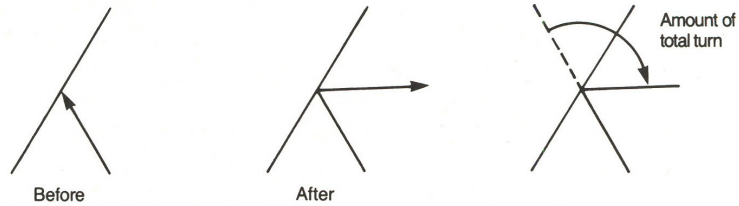
$$(line's\ heading) - (turtle's\ heading)$$

as the angle of incidence. The angle of reflection should also be

$$(line's\ heading) - (turtle's\ heading)$$

The total amount through which the turtle turns is therefore

$$2 \times \{(line's\ heading) - (turtle's\ heading)\}$$



Figuring Out the Heading

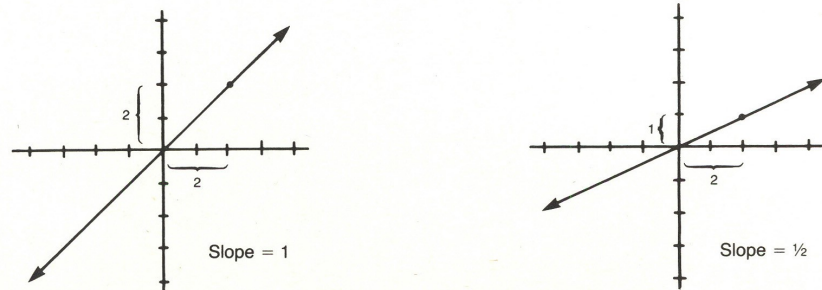
In fact, we are not given the heading or the position of a line. All we are given are its two endpoints. With the two endpoints we can figure out the slope. Then we can use the ARCTAN procedure, described in the Towards and Arctan project, to figure out the heading. The procedure to figure out the heading is as follows.

```
TO FIGH :LINE
IF ( - DX :LINE ) = 0 [OP 0]
OP 90 - ARCTAN SLOPE :LINE
END
```

(It is 90-ARCTAN because Logo headings are clockwise from north, not counterclockwise from east as in algebra.)

Figuring Out the Slope

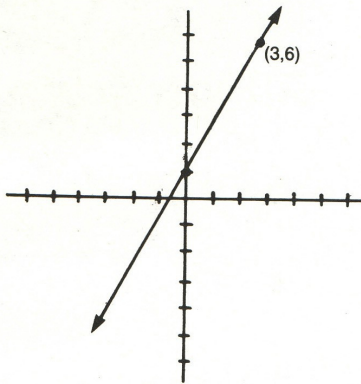
The slope is the difference between the y coordinates of any two points on the line divided by the difference between the x coordinates of those points ($\Delta y / \Delta x$, where Δ stands for "difference in").



Traditionally, a line is represented by the equation $y = mx + b$. Given m and b , we can tell whether a particular point is on a particular line. For example, if

$$YCOR = (m \times XCOR) + b$$

then we know that the turtle's position is on the line.



m of line = $\frac{5}{3}$ $(\frac{5}{3} * 3) + 1 = 6$
 b of line = 1 So the point is
 $YCOR = 6$ on the line.
 $XCOR = 3$

We use $\Delta y / \Delta x$ to figure out the slope. But if the line is vertical (the two x coordinates are the same), then the slope is infinite, so the procedure to figure out the slope has to treat that case in a special way. The procedures to figure out the slope (m) and y -intercept (b) of the line are as follows.

```

TO FIGM :LINE
  IF ( DX :LINE ) = 0 [OP []] [OP SLOPE :LINE]
END

TO SLOPE :LINE
  OP ( DY :LINE ) / DX :LINE
END

TO DY :LINE
  OP ( LAST POINT1 :LINE ) - ( LAST POINT2 :LINE )
END

TO DX :LINE
  OP ( FIRST POINT1 :LINE ) - ( FIRST POINT2 :LINE )
END

TO FIGB :LINE
  IF ( M :LINE ) = [] [OP []]
  OP ( LAST POINT1 :LINE ) - ((M :LINE) * (FIRST POINT1 :LINE))
END

```

Information We Need About a Line

What information does this program need about a line?

It needs the heading for use in calculating the turning angle when the turtle hits the line.

PROGRAMMING IDEAS

It uses slope and y -intercept to figure out if a position is on a line, with the equation $y = mx + b$.

It also needs the endpoints. Why? So far we have been talking about *lines*, which are infinitely long. Really the program has to deal with *line segments*, which have two endpoints.

So we finally need five pieces of information to represent a line segment: two endpoints, heading, slope, and y -intercept.

Storing the Lines

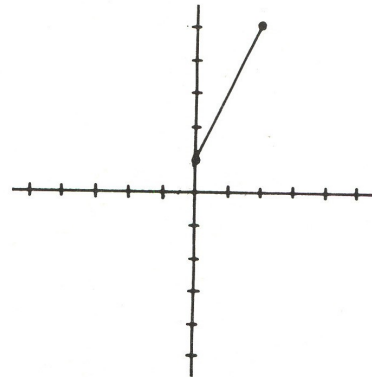
What Each Line Looks Like

The only thing that the user gives the program is the endpoints of lines. From this the slope, heading, and y -intercept are figured out. The program has to have a way of storing all this information in some kind of organized structure. It stores the five pieces of information about the line in a list of the following format:

```
[[x1 y1] [x2 y2] slope heading y-intercept]
```

Here is a sample line and the list that represents it.

```
[[0 1] [2 5] 2 26.5651 1]
```



Retrieving Information About Lines

Throughout the program we need to recall information about lines. We could do it in a messy way. For example, to retrieve the slope of a line we could say

```
FIRST BF BF :LINE
```

The program would get very ugly and confusing if we used that approach. A much clearer and neater way is to have a procedure that extracts one piece of information about a line. So we could say `M :LINE` to retrieve the slope, or `POINT2 :LINE` to retrieve the coordinates of the second endpoint. By using these procedures, other parts of the program do not have to know the detailed structure of a line list. The procedures also make

the program much easier to read and understand. Here are the information retrieving procedures.

```
TO POINT1 :LINE
OP ITEM 1 :LINE
END
```

```
TO POINT2 :LINE
OP ITEM 2 :LINE
END
```

```
TO M :LINE
OP ITEM 3 :LINE
END
```

```
TO D :LINE
OP ITEM 4 :LINE
END
```

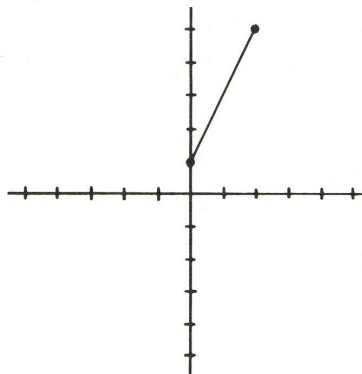
```
TO B :LINE
OP ITEM 5 :LINE
END
```

```
TO ITEM :INUM :LIST
IF :INUM = 1 [OP FIRST :LIST]
OP ITEM :INUM - 1 BUTFIRST :LIST
END
```

The List of Lines

Since the program has to keep track of a lot of lines, it stores them all in a list called `LINES`. The elements of `:LINES` are themselves lists, each representing a line. For example, the border lines and the line shown earlier would be represented as follows:

```
[ [[-158 -119] [-158 120] [] 0 []]
  [[161 -119] [161 120] [] 0 []]
  [[-158 120] [161 120] 0 90 120]
  [[-158 -119] [161 -119] 0 90 -119]
  [[0 1] [2 5] 2 26.5651 1] ]
```



Program Structure

The top-level procedure is BOUNCE. It has four tasks. First it creates the list of lines. Then it sets up the initial position and shape of the turtle. The third task is to draw the lines in the list. Finally it starts the turtle moving and prepares to turn the turtle when it bounces off a wall. There is a subprocedure for each of these tasks.

```
TO BOUNCE
  LEARN.LINES
  SETUP.GRAPHICS
  DRAW.LINES
  START.TURTLE
END
```

Creating the List of Lines

When the program starts up, LEARN.LINES creates the list of lines. It calls INFO, to remember the lines which the user enters. It also calls BORDER, which remembers the border lines.

```
TO LEARN.LINES
  MAKE "LINES []
  INFO
  BORDER
END
```

INFO lets the user enter lines. It calls GETLINE to get each line and calls REMEMBER to add each line to the list :LINES.

```
TO INFO
  REMEMBER GETLINE
  PRINT [ANOTHER LINE?]
  IF EQUALP RL [YES] [INFO]
END
```

GETLINE asks the user to type in the endpoints of a line segment. It calls FIGLINE to calculate the other information about the line. The output from GETLINE is the list representing the line.

```
TO GETLINE
  TYPE [X AND Y OF FIRST POINT?]
  MAKE "FP RL
  TYPE [X AND Y OF SECOND POINT?]
  MAKE "SP RL
  MAKE "LINE LIST :FP :SP
  OP FIGLINE :LINE
END
```

FIGLINE takes the endpoints of a line segment as its input. It calls FIGM, FIGH, and FIGB to compute the slope, heading, and *y*-intercept of the line.

```

TO FIGLINE :LINE
MAKE "LINE LPUT FIGM :LINE :LINE
MAKE "LINE LPUT FIGH :LINE :LINE
MAKE "LINE LPUT FIGB :LINE :LINE
OP :LINE
END

```

REMEMBER adds a line to the list of lines (:LINES).

```

TO REMEMBER :LINE
MAKE "LINES FPUT :LINE :LINES
END

```

BORDER remembers the four lines making up the border of the screen.

```

TO BORDER
REMEMBER FIGLINE [[-158 -119] [-158 120]]
REMEMBER FIGLINE [[161 -119] [161 120]]
REMEMBER FIGLINE [[-158 120] [161 120]]
REMEMBER FIGLINE [[-158 -119] [161 -119]]
END

```

Setting Up Graphics

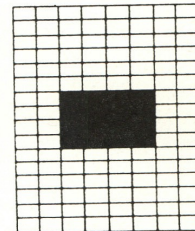
SETUP.GRAPHICS selects turtle 0 and changes its shape. We don't use the normal turtle shape because later on we will need to know the precise position of the turtle. The normal turtle shape is big enough that its edges are at a very different position from the position of the center, which XCOR and YCOR output. Instead, we use a small square dot shape.

```

TO SETUP.GRAPHICS
ASK [1 2 3] [HT]
TELL 0
PUTSH 1 :SHAPE1
SETSH 1
CS
ST
FS
END

```

```
MAKE "SHAPE1 [0 0 0 0 0 0 60 60 60 60 0 0 0 0 0 0]
```



Drawing the Lines

DRAW.LINES draws the lines in :LINES on the screen. The lines are drawn with pen 1. Later, when the turtle is moving, its trajectory is drawn with pen 0. Using a different pen for the walls allows the demon to notice collisions with the walls and not notice collisions between the turtle and its own earlier path.

```

TO DRAW.LINES
SETPN 1
DRAW :LINES
END

```

PROGRAMMING IDEAS

```

TO DRAW :LIST
IF EMPTY? :LIST [STOP]
PU
SETPOS POINT1 FIRST :LIST
PD
SETPOS POINT2 FIRST :LIST
DRAW BUTFIRST :LIST
END

```

Starting the Turtle

START.TURTLE positions the turtle in the center of the screen, points it in a randomly chosen direction, and starts it moving. It also creates the demon that waits for collisions with lines. Finally, START.TURTLE calls LOOP, which is explained next.

```

TO START.TURTLE
SETPN 0
PU
SETPOS [0 0]
PD
SETH RANDOM 360
SETSP 20
WHEN OVER 0 1 [SETSP 0 WHEN OVER 0 1 []]
LOOP
END

```

Knowing When a Line Is Hit

While the turtle is moving, LOOP continually checks if it has hit a line. LOOP knows the turtle has hit a line when its speed becomes zero.

```

TO LOOP
IF SPEED = 0 [NEWHEAD]
LOOP
END

```

How does the speed become zero? There is a demon, created by START.TURTLE, whose instructions include SETSP 0.

```

WHEN OVER 0 1 [SETSP 0 WHEN OVER 0 1 []]

```

Setting the turtle's speed to zero is a convenient way for the demon to signal to LOOP that the turtle has hit a line. We could have changed something else as the signal, but we had to stop the turtle anyway. Otherwise the turtle would go through the line. Using the speed as the signal solves two problems at once.

When the speed is zero, LOOP calls NEWHEAD to figure out which line was hit and how much the turtle should turn.

Which Line Is Being Hit?

When a line is hit, NEWHEAD calls SEARCH to go through the list of lines, finding the one that was hit. Then NEWHEAD uses that line as the input to FIGTURN, which figures out how much the turtle should turn. Finally, NEWHEAD restarts the turtle and the demon.

```
TO NEWHEAD
MAKE "L SEARCH :LINES
IF NOT EMPTY? L [RIGHT FIGTURN :L]
SETSP 20
WHEN OVER 0 1 [SETSP 0 WHEN OVER 0 1 []]
END
```

SEARCH goes through the list of lines, looking for the one the turtle hit. It calls the predicate CHECK for each line in the list. If CHECK outputs TRUE, SEARCH outputs the line that has been found.

```
TO SEARCH :LINES
IF EMPTY? :LINES [OP []]
IF CHECK FIRST :LINES [OP FIRST :LINES]
OP SEARCH BF :LINES
END
```

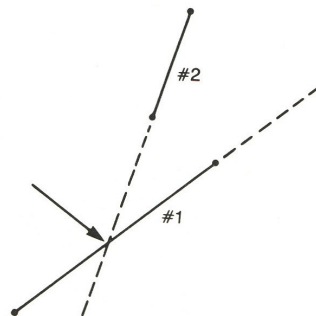
How to Check a Line

The straightforward way to check if the turtle hit a certain line is to use the equation $y = mx + b$, substituting the XCOR and YCOR of the turtle for x and y . If the equation holds true, then the turtle hit that line.

```
TO CHECK1 :LINE
OP YCOR = SOLVE :LINE XCOR
END

TO SOLVE :LINE :X
OP ( ( M :LINE ) * :X ) + B :LINE
END
```

There are three problems. The first problem has to do with the fact that we are using line *segments* and not lines. Look at the following picture.



PROGRAMMING IDEAS

The turtle has hit line segment #1. The turtle's coordinates satisfy the equation $y = mx + b$ for the line containing that segment. The turtle has *not* hit line segment #2. However, the line containing that segment happens to pass through the turtle's position. Therefore, the equation $y = mx + b$ for *that* line is also satisfied. CHECK must also check to see if the turtle is *between* the two endpoints of the line segment.

```

TO CHECK2 :LINE
  IF NOT BETWEEN XCOR (FIRST POINT1 :LINE)
    (FIRST POINT2 :LINE) [OP FALSE]
  OP YCOR = SOLVE :LINE XCOR
END

TO BETWEEN :THING :LIMIT1 :LIMIT2
  IF :LIMIT1 > :LIMIT2 [OP AND ( GE :LIMIT1 :THING )
    ( GE :THING :LIMIT2 ) ]
    [OP AND ( GE :LIMIT2 :THING ) ( GE :THING :LIMIT1 ) ]
END

TO GE :A :B
  OP NOT :A < :B
END

```

The second problem is that the turtle isn't actually one point; it is slightly bigger. This means that when the edge of the turtle hits a line, the turtle's coordinates won't match up exactly with the line's, because the turtle's coordinates are those of its center, not those of its edge. In this program the turtle has a square shape. If its YCOR is within seven units of the line's y value for the turtle's XCOR, we consider the turtle to be on the line. The number seven worked out best experimentally. Larger numbers lead to false hits. Smaller ones lead to not finding any hits at all. Our updated version of CHECK looks like this.

```

TO CHECK3 :LINE
  IF NOT BETWEEN XCOR (FIRST POINT1 :LINE)
    (FIRST POINT2 :LINE) [OP FALSE]
  OP ( ABS (SOLVE :LINE XCOR) - YCOR ) < 7
END

TO ABS :NUMBER
  OP IF :NUMBER < 0 [- :NUMBER] [:NUMBER]
END

```

The last problem is that vertical lines have an infinite slope (Δx is zero in $\Delta y / \Delta x$). SOLVE won't work for a vertical line, because it needs a numeric slope. Also, we can't check to see if the turtle is between the x values of the endpoints of the line, because the x values are the same; there is no "be-

tween.”* So, for a vertical segment, we have to see if the YCOR of the turtle is between the y values of the two endpoints.

Instead of calling SOLVE, we see if the XCOR of the turtle is within seven units of the x value of one of the endpoints. Our final version of CHECK looks like this.

```
TO CHECK :LINE
OP IF EMPTY M :LINE [CHECK.VERT :LINE] [CHECK.SLANT :LINE]
END

TO CHECK.SLANT :LINE
IF NOT BETWEEN XCOR ( FIRST POINT1 :LINE )
  ( FIRST POINT2 :LINE ) [OP "FALSE]
OP ( ABS ( SOLVE :LINE XCOR ) - YCOR ) < 7
END

TO CHECK.VERT :LINE
IF NOT BETWEEN YCOR ( LAST POINT1 :LINE )
  ( LAST POINT2 :LINE ) [OP "FALSE]
OP ( ABS XCOR - FIRST POINT1 :LINE ) < 7
END
```

Turning the Turtle

Once NEWHEAD knows which line was hit, it can figure out how much to turn the turtle. The turtle's original heading is provided by the primitive procedure HEADING. The heading of the line is provided by

```
D :LINE
```

Recall that the angle through which the turtle should turn is therefore

```
2 * ( ( D :LINE ) - HEADING )
```

NEWHEAD calls FIGTURN to figure out how much the turtle should turn:

```
TO FIGTURN :L
OP 2 * ( ( D :L ) - HEADING )
END
```

*This is not exactly true. If the turtle's XCOR is *equal* to the x values of the line, then in a sense the turtle is between the x values. This doesn't mean the turtle is on the line segment. The problem is that for a vertical line segment, the turtle's YCOR might not be between the y values of the endpoints of the line segment, even though the XCOR is in the right range. For diagonal lines, if one coordinate is in range, the other must also be in range.

PROGRAM LISTING

```

TO FIGH :LINE
IF ( DX :LINE ) = 0 [OP 0]
OP 90 - ARCTAN SLOPE :LINE
END

TO FIGM :LINE
IF ( DX :LINE ) = 0 [OP []] [OP SLOPE ►
:LINE]
END

TO SLOPE :LINE
OP ( DY :LINE ) / DX :LINE
END

TO DY :LINE
OP ( LAST POINT1 :LINE ) - ( LAST ►
POINT2 :LINE )
END

TO DX :LINE
OP ( FIRST POINT1 :LINE ) - ( FIRST ►
POINT2 :LINE )
END

TO FIGB :LINE
IF ( M :LINE ) = [] [OP []]
OP ( LAST POINT1 :LINE ) - ( ( M :LINE ) ►
* ( FIRST POINT1 :LINE ) )
END

TO POINT1 :LINE
OP ITEM 1 :LINE
END

TO POINT2 :LINE
OP ITEM 2 :LINE
END

TO M :LINE
OP ITEM 3 :LINE
END

TO D :LINE
OP ITEM 4 :LINE
END

TO B :LINE
OP ITEM 5 :LINE
END

TO ITEM :INUM :LIST
IF :INUM = 1 [OP FIRST :LIST]

OP ITEM :INUM - 1 BUTFIRST :LIST
END

TO BOUNCE
LEARN.LINES
SETUP.GRAPHICS
DRAW.LINES
START.TURTLE
END

TO LEARN.LINES
MAKE "LINES []
INFO
BORDER
END

TO INFO
REMEMBER GETLINE
PRINT [ANOTHER LINE?]
IF EQUALP RL [YES] [INFO]
END

TO GETLINE
TYPE [X AND Y OF FIRST POINT?]
MAKE "FP RL
TYPE [X AND Y OF SECOND POINT?]
MAKE "SP RL
MAKE "LINE LIST :FP :SP
OP FIGLINE :LINE
END

TO FIGLINE :LINE
MAKE "LINE LPUT FIGM :LINE :LINE
MAKE "LINE LPUT FIGH :LINE :LINE
MAKE "LINE LPUT FIGB :LINE :LINE
OP :LINE
END

TO REMEMBER :LINE
MAKE "LINES FPUT :LINE :LINES
END

TO BORDER
REMEMBER FIGLINE [[-158 -119] [-158 ►
120]]
REMEMBER FIGLINE [[161 -119] [161 ►
120]]
REMEMBER FIGLINE [[-158 120] [161 ►
120]]
REMEMBER FIGLINE [[-158 -119] [161 ►
-119]]
END

```

```

TO SETUP.GRAPHICS
ASK [1 2 3] [HT]
TELL 0
PUTSH 1 :SHAPE1
SETSH 1
CS
ST
FS
END

```

```

TO DRAW.LINES
SETPN 1
DRAW :LINES
END

```

```

TO DRAW :LIST
IF EMPTY? :LIST [STOP]
PU
SETPOS POINT1 FIRST :LIST
PD
SETPOS POINT2 FIRST :LIST
DRAW BUTFIRST :LIST
END

```

```

TO START.TURTLE
SETPN 0
PU
SETPOS [0 0]
PD
SETH RANDOM 360
SETSP 20
WHEN OVER 0.1 [SETSP 0 WHEN OVER 0.1 ►
[]]
LOOP
END

```

```

TO LOOP
IF SPEED = 0 [NEWHEAD]
LOOP
END

```

```

TO NEWHEAD
MAKE "L SEARCH :LINES
IF NOT EMPTY? :L [RIGHT FIGTURN :L]
SETSP 20
WHEN OVER 0.1 [SETSP 0 WHEN OVER 0.1 ►
[]]
END

```

```

TO SEARCH :LINES
IF EMPTY? :LINES [OP []]
IF CHECK FIRST :LINES [OP FIRST ►
:LINES]
OP SEARCH BF :LINES
END

```

```

TO SOLVE :LINE :X
OP ( ( M :LINE ) * :X ) + B :LINE
END

```

```

TO BETWEEN :THING :LIMIT1 :LIMIT2
IF :LIMIT1 > :LIMIT2 [OP AND ( GE ►
:LIMIT1 :THING ) ( GE :THING ►
:LIMIT2 )] [OP AND ( GE :LIMIT2 ►
:THING ) ( GE :THING :LIMIT1 )]
END

```

```

TO GE :A :B
OP NOT :A < :B
END

```

```

TO ABS :NUMBER
OP IF :NUMBER < 0 [- :NUMBER] ►
[:NUMBER]
END

```

```

TO CHECK :LINE
OP IF EMPTY? M :LINE [CHECK.VERT ►
:LINE] [CHECK.SLANT :LINE]
END

```

```

TO CHECK.SLANT :LINE
IF NOT BETWEEN XCOR ( FIRST POINT1 ►
:LINE ) ( FIRST POINT2 :LINE ) ►
[OP "FALSE]
OP ( ABS ( SOLVE :LINE XCOR ) - YCOR ) ►
< 7
END

```

```

TO CHECK.VERT :LINE
IF NOT BETWEEN YCOR ( LAST POINT1 ►
:LINE ) ( LAST POINT2 :LINE ) [OP ►
"FALSE]
OP ( ABS XCOR - FIRST POINT1 :LINE ) < ►
7
END

```

```

TO FIGTURN :L
OP 2 * ( ( D :L ) - HEADING )
END

```

```

TO ARCTAN :X
OP 57.3*ARCTAN.RAD :X
END

```

```

TO ARCTAN.RAD :X
IF :X>1 [OP 1.571-ARCTAN.RAD (1/:X)]
OP :X/(1+0.28*:X*:X)
END

```

```

MAKE "SHAPE1 [0 0 0 0 0 0 60 60 60 60 ►
0 0 0 0 0]

```